Google Android

Crie aplicações para celulares e tablets

JOÃO BOSCO MONTEIRO

AGRADECIMENTOS

Gostaria primeiramente de agradecer a você, leitor. Este livro foi preparado especialmente para que você adquira rapidamente os conhecimentos necessários, por meio de exemplos práticos, para desenvolver aplicativos para a plataforma Android.

Agradeço, em especial, ao Adriano Almeida e ao Paulo Silveira a oportunidade de escrever este livro e também a todas as pessoas que contribuíram para que isto fosse possível, incluindo os amigos que ajudaram na revisão e também minha esposa, pela compreensão e incentivo.

Sumário

Versão: 19.5.20

Capítulo 1

CONSTRUA SUA PRIMEIRA APLICAÇÃO

Com o passar do tempo, os telefones celulares foram evoluindo, ganhando cada vez mais recursos e tornando-se um item quase indispensável na vida das pessoas. E não foi apenas isso que mudou. Também houve uma mudança significativa para nós, os desenvolvedores de software.

Antes, o mercado de desenvolvimento para celulares era praticamente restrito aos fabricantes e operadoras que controlavam a criação e inclusão dos aplicativos em seus aparelhos. A liberação, por parte dos fabricantes, de um kit de desenvolvimento de software (SDK) para suas plataformas e a criação de lojas para a distribuição de aplicativos viabilizaram a abertura deste mercado para praticamente qualquer empresa ou desenvolvedor, criando, assim, novas oportunidades de negócio.

A plataforma Android desfruta hoje de um papel de destaque no mercado tanto pela quantidade significativa de dispositivos produzidos como também por oferecer uma API rica, disponibilizando fácil acesso a vários recursos de hardware, tais como Wi-Fi e GPS, além de boas ferramentas para o desenvolvedor. A facilidade de desenvolver utilizando uma linguagem de programação (Java) bastante disseminada, a simplicidade e o baixo custo para a publicação de aplicativos na loja Google Play e a quantidade de dispositivos Android em uso no mundo só fazem

aumentar a popularidade da plataforma.

Segundo o relatório do *International Data Corporation* (IDC) publicado em maio de 2012, o Android possui 59% do mercado de smartphones e soma a quantia de 89,9 milhões de aparelhos distribuídos apenas no primeiro trimestre deste ano (2012), em todo o mundo. Em segundo lugar, aparece o iOS que é o sistema operacional do Apple iPhone. O gráfico a seguir demonstra a participação no mercado dos principais sistemas operacionais e a quantidade de aparelhos distribuídos.
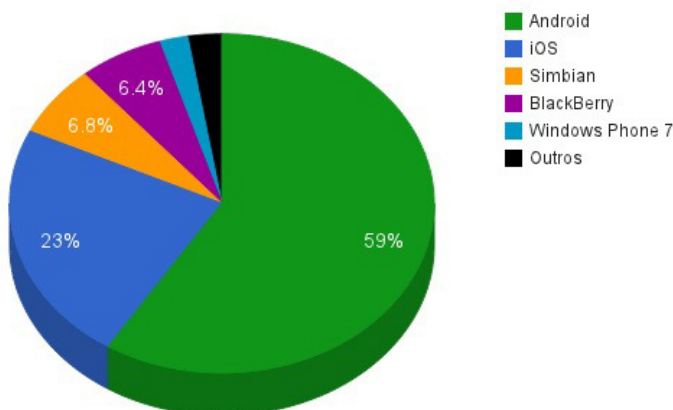
Figura 1.1: Participação no mercado. Fonte: IDC

O IDC também prevê que, em 2016, o Android ainda possuirá a maior fatia do mercado, com 52,9%. A disputa pelo segundo lugar será acirrada entre iOS e Windows Phone 7. O gráfico a seguir ilustra a previsão realizada pelo IDC.
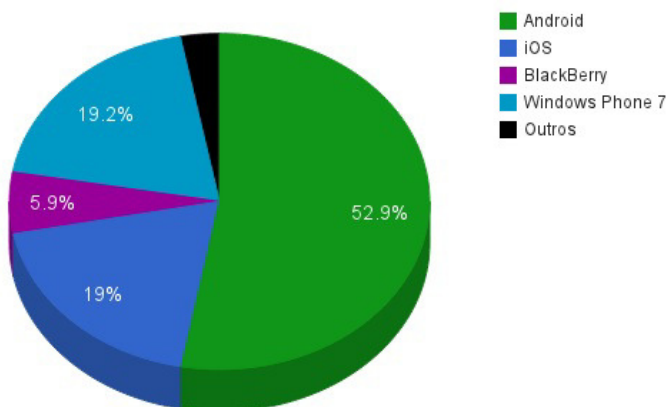
Figura 1.2: Previsão do mercado para 2016. Fonte: IDC

Já no mercado de tablets, o iPad detém o trono com 68%, enquanto o Google aposta em seu primeiro tablet, o Nexus 7 ao preço US$199,00, para ganhar terreno tentando repetir o sucesso do Amazon Kindle Fire, que também utiliza Android.

1.1 CONHEÇA O ANDROID

Desenvolvido especialmente para dispositivos móveis, como aparelhos celulares e tablets, e agora também suportando dispositivos "vestíveis" (*wearables*), como relógios, o Android é uma plataforma composta por um sistema operacional, *middlewares* e um conjunto de aplicativos principais, como os contatos, o navegador de internet e o telefone propriamente dito. Além disso, existe o Android SDK, que é um conjunto de ferramentas e APIs para o desenvolvimento de aplicativos para a plataforma, utilizando a linguagem Java.

No decorrer deste livro, vamos abordar em detalhes os componentes existentes no Android, além de mostrar como integrá-los para criar aplicações ricas em funcionalidades e com uma

usabilidade agradável.

Baseado no Linux, o sistema operacional Android teve seu desenvolvimento iniciado em 2003 pela empresa Android Inc. Em 2005, esta foi adquirida pelo Google, que hoje lidera o desenvolvimento do Android. Um marco importante dessa trajetória aconteceu em 2007, com a criação da Open Handset Alliance (http://www.openhandsetalliance.com/), que é uma associação de empresas de software, hardware e telecomunicações, cuja missão é desenvolver uma plataforma para dispositivos móveis que seja completa, aberta e gratuita. Também em 2007 ocorreu o lançamento da versão beta do primeiro SDK para Android. Após diversas versões e melhorias, em outubro de 2014 foi lançado o Android 5.0, codinome *Lollipop*, e em maio de 2015 o `developer preview` da próxima versão, conhecida como M Preview, foi liberada.

UM POUCO MAIS DE HISTÓRIA

Quer saber mais sobre a história do Android, suas versões e evoluções? Visite http://www.xcubelabs.com/the-android-story.php e http://www.theverge.com/2011/12/7/2585779/android-history.

As grandes novidades da versão 5.0 do Android ficam por conta da nova interface gráfica *Material Design*, focada em fornecer para o usuário uma experiência de uso consistente, responsiva e intuitiva, entre vários tipos de dispositivos. Há também melhorias de conectividade com destaque para o suporte melhorado ao *Bluetooth Low Energy* (BLE). O novo *Android Runtime* (ART) contribui para uma melhoria significativa de desempenho, tornando a execução das aplicações mais rápidas.

Outras melhorias e evoluções foram introduzidas, tais como: nova funcionalidade de economia de bateria; novas funções de segurança, como encriptação automática de dados; possibilidade de criação de perfis de acesso para compartilhar o seu telefone, mas proteger os seus dados e suas informações privadas; acesso facilitado ao recurso OK Google, mesmo com a tela desligada; suporte ao Android TV e muito mais.

ANDROID É CÓDIGO ABERTO!

O Android é código aberto e distribuído sob licença Apache 2.0, o que quer dizer que você tem acesso aos códigos-fontes e também pode contribuir com o projeto! Saiba mais em http://source.android.com.

1.2 CONFIGURE O AMBIENTE PARA DESENVOLVIMENTO

Antes de criar a nossa primeira aplicação Android, é necessário baixar e instalar o Android SDK e a IDE (*Integrated Development Environment*) para auxiliar o desenvolvimento. O Google disponibilizou oficialmente no final de 2014 um novo conjunto de ferramentas chamado de **Android Studio**, que já contém tudo o que é necessário para começar a criar aplicativos. Ele está disponível no endereço http://developer.android.com/sdk. Escolha o pacote mais adequado de acordo com o seu sistema operacional, faça o download e, para instalar, basta seguir as instruções do instalador. Lembre-se que é necessário possuir um JDK 7 ou superior instalado.

O Android Studio conta com o Android Studio IDE — uma IDE baseada no IntelliJ IDEA —, o Android SDK — que reúne as

ferramentas necessárias para compilar e emular as aplicações —, a última versão da plataforma Android e também uma imagem de sistema para o emulador. No diretório que foi instalado, haverá o diretório `sdk`, onde ficam as ferramentas, bibliotecas e APIs do Android, organizadas por versão. Agora que temos o ambiente de desenvolvimento preparado, estamos prontos para escrever nossa primeira aplicação Android!

Como uma primeira experiência em uma nova plataforma, vamos desenvolver um clássico `Hello World`. No Android Studio, devemos criar um novo projeto Android, por meio da opção `Start a New Android Studio project`, disponível na tela principal. A figura seguinte destaca a opção que deve ser selecionada. Esta tela principal será exibida sempre que o Android Studio for aberto.
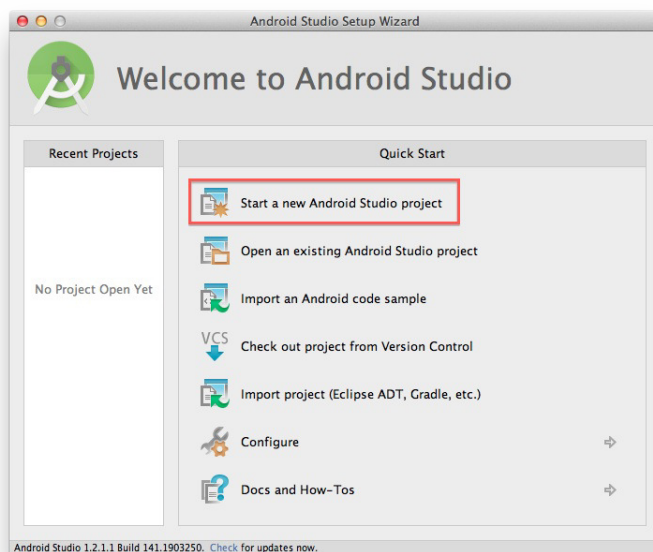
Figura 1.3: Iniciando um novo projeto Android

Na tela seguinte, escolha um nome para o seu projeto; no nosso

caso, vamos chamar de `HelloAndroid` . É importante escolher o nome da aplicação e do pacote com cautela, pois esses dois nomes serão utilizados para identificar sua aplicação quando for feita uma publicação no Google Play. A recomendação é utilizar o nome do pacote como sendo algum domínio seu ou da sua empresa somado ao nome da aplicação. Para o nosso exemplo, escolhemos o nome `HelloAndroid` (no caso, mesmo nome do projeto) e o pacote `br.com.casadocodigo.helloandroid` . Ao informar o nome do projeto e o domínio da empresa, o próprio Android Studio já sugere o nome do pacote. Veja como ficou a configuração na figura:
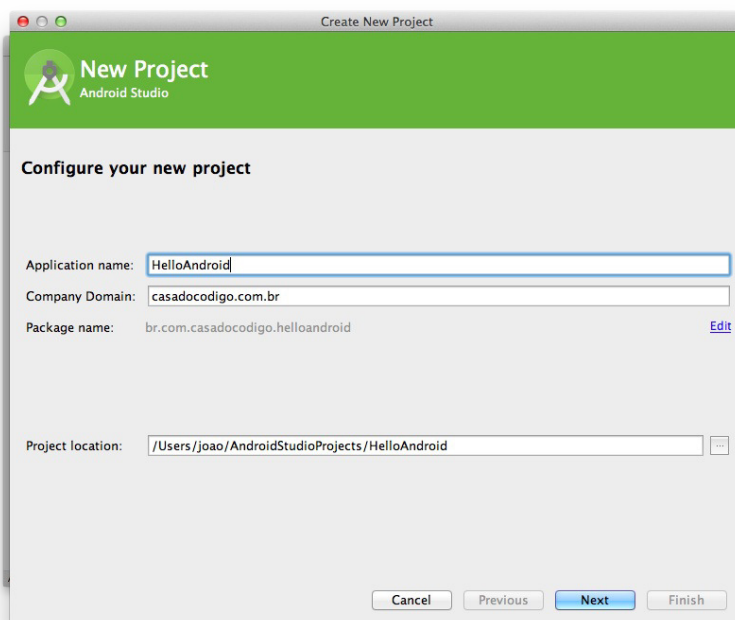
Figura 1.4: Configurando o nome e pacote da aplicação

Ao prosseguir para a próxima tela, é necessário informar para quais tipos de dispositivos estamos desenvolvendo nossa aplicação. A opções disponíveis são: `Phone and Tablet` , `TV` e `Wear` .

Utilizaremos a opção para telefones e tablets, informando a versão mínima necessária. A opção `Minimum SDK` indica qual é a versão mínima exigida para executar o aplicativo. Vamos escolher a versão Android 2.3.3 (API 10), pois usaremos recursos que estão disponíveis apenas a partir desta, como mostra a figura a seguir. O Android Studio traz um guia para ajudar a escolher a versão mais adequada do Android.
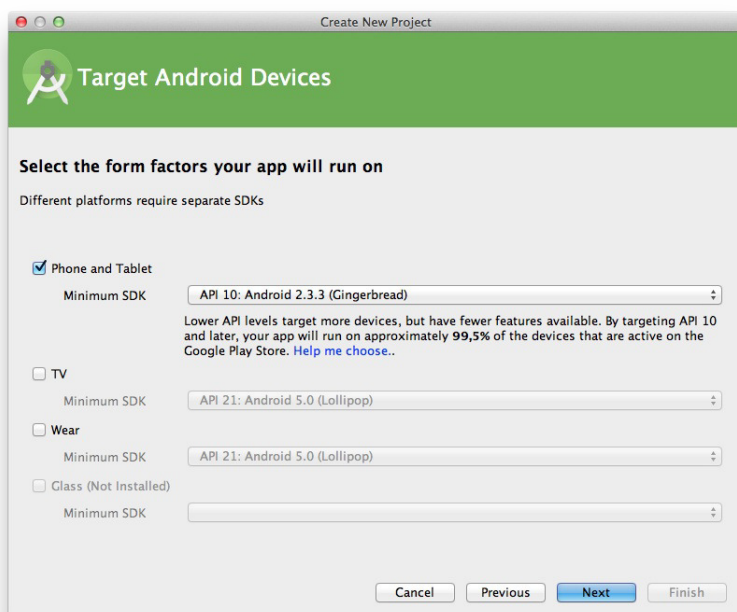
Figura 1.5: Escolhendo a versão do SDK

Escolhendo a opção `Help me choose`, uma nova janela é apresentada, exibindo um infográfico com as principais versões do Android e o percentual da base instalada de cada uma delas. A escolha de qual versão suportar deve ser balizada por esses valores, bem como pelos recursos existentes e que se desejam utilizar de uma determinada versão. A figura a seguir mostra, por exemplo, que, ao

utilizar a versão 2.3.3, estamos atingindo aproximadamente 99,5% dos dispositivos Android.
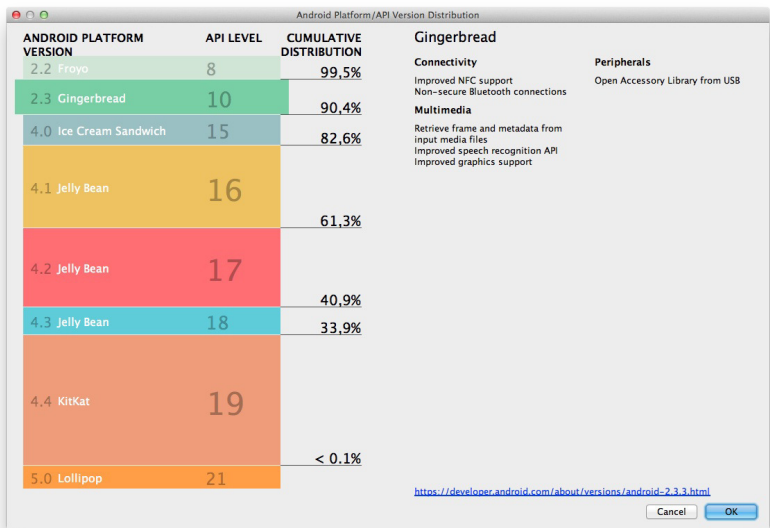
Figura 1.6: Escolhendo a versão do Android

Prossiga para o próximo passo clicando em `Next`. A próxima tela apresenta algumas opções de template para a criação de `Activities`, no entanto não utilizaremos nenhuma agora. Usar um template ajuda a agilizar a criação do projeto com as estruturas mais utilizadas, mas, no momento, precisamos nos concentrar no aprendizado. Depois de dominar os fundamentos, você poderá facilmente utilizar qualquer um dos templates. Na tela apresentada, escolha a opção `Add no Activity`, como mostra a figura seguinte. Selecione a opção `Finish` para encerrar o assistente de criação do projeto. Ao final desse processo, teremos um novo projeto Android criado.
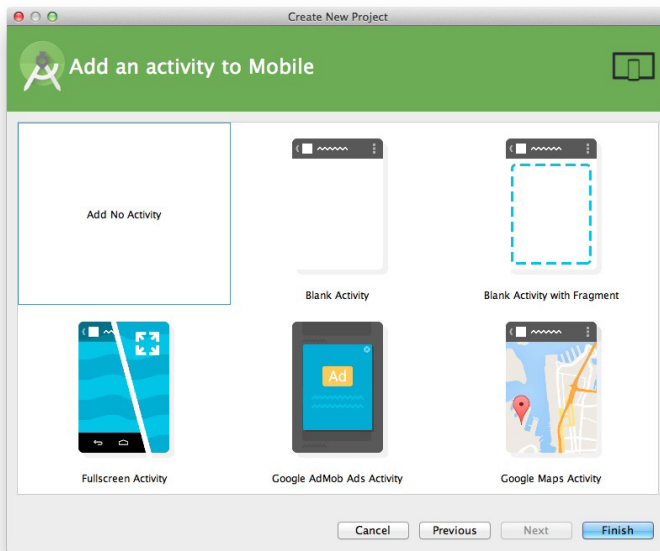
Figura 1.7: Escolhendo uma template de projeto

Para implementar o nosso `Hello World`, precisaremos criar uma classe Java e um arquivo `XML` com a definição do layout da tela. Certifique-se de que a visão Android esteja selecionada no Android Studio, como mostra a figura adiante. Começaremos criando um novo `XML` de layout, utilizando a opção `File > New > Android XML File`. No campo `File`, informe o nome do arquivo como `activity_main`. Não é necessário alterar as demais informações. Ao finalizar, o Android Studio abrirá o arquivo que terá o conteúdo semelhante ao mostrado no código a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</LinearLayout>
```
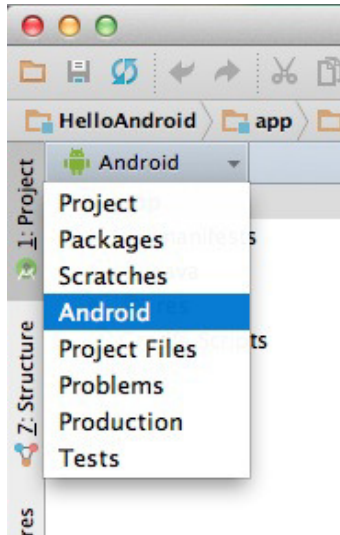
Figura 1.8: Tipos de visão de um projeto no Android Studio

Neste arquivo de definição de layout, temos a declaração de um elemento `LinearLayout` e seus respectivos atributos. Ele é empregado para organização do layout da tela, permitindo configurar a sua altura e largura, além de acomodar outros componentes de acordo com uma orientação (vertical ou horizontal). Agora precisamos incluir outro elemento, que será o responsável por exibir uma informação textual para o usuário. Para isto, usaremos um `TextView` . O valor a ser exibido por este elemento é especificado por meio do atributo `text` . Veja como fica o código do arquivo de layout após a inclusão do `TextView` .

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
```

```
</LinearLayout>
```

Repare que, na linha 10, o valor que `TextView` deve exibir é `@string/hello_world`. O valor que será utilizado no `TextView` será, na verdade, o conteúdo da `string` que possui o identificador `hello_world`. Esta é a forma de externalizar `strings` no Android e evitar que o layout e o código mantenham `strings` específicas, o que pode dificultar a manutenção do aplicativo e também a disponibilização deste para outros idiomas. A definição das `strings` deve ser realizada no arquivo `res/values/strings.xml`, como pode ser visto no código a seguir.

```
<resources>
    <string name="app_name">HelloAndroid</string>
    <string name="hello_world">Hello World!</string>
</resources>
```

Declaramos uma `string` com o nome `hello_world`, cujo valor é `Hello World!`. Por convenção, o arquivo `strings.xml` é onde definimos recursos do tipo `string`, ou seja, textos que queremos exibir de alguma maneira em nossa aplicação. Prosseguindo com o exemplo, precisamos criar uma classe Java representando uma atividade (`Activity`) que exibirá essa tela.

As *activities*, ou atividades, são componentes da plataforma Android, capazes de apresentar uma tela para interagir com os usuários. Por meio delas, podemos tirar uma foto, enviar um e-mail, visualizar uma imagem e navegar na internet. Geralmente, uma aplicação é composta por várias *activities*, sendo uma delas a `activity` principal, que é executada quando a iniciamos.
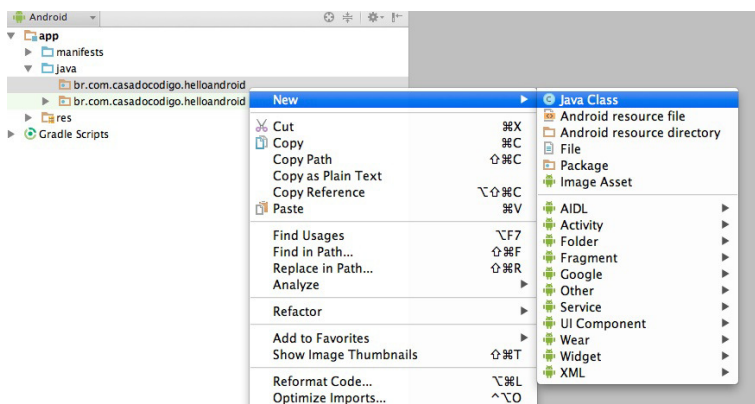
Figura 1.9: Criar uma classe no Android Studio

Para criar uma nova classe, clique com o botão direito em cima do pacote (mostrado na figura anterior), selecione a opção `New > Java Class` e informe `MainActivity` como nome da classe. Para criar uma atividade, basta fazer com que nossa classe estenda a classe `Activity` do Android. Depois precisamos sobrescrever o método `onCreate` para informar qual é o arquivo de layout utilizado pela `activity`. O código a seguir demonstra a classe `MainActivity` do nosso projeto HelloWorld:

```
public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

A linha 1 mostra a herança da classe `Activity`. Já na linha 6, passamos para o método `setContentView` o identificador do layout, `R.layout.activity_main`, que deve ser carregado para construir a interface gráfica da nossa atividade. O último passo para tornar esse projeto funcional é informar no arquivo `AndroidManifest.xml`, localizado no diretório `manifests` na

visão Android, qual deve ser a `activity` que deve ser executada quando o aplicativo for aberto. Para isso, acrescente um elemento `activity` como parte da aplicação (tag `application`), como mostra o código a seguir:

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.casadocodigo.helloandroid">

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity" >
        <intent-filter>
          <action
             android:name="android.intent.action.MAIN" />
          <category
             android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
</application>

</manifest>
```

Não se preocupe ainda com os detalhes dessa configuração. Vamos rodar nosso primeiro aplicativo! Selecione o menu `Run` > `Run` ; uma caixa de diálogo `Run as` vai aparecer solicitando que seja escolhido um dispositivo virtual para executá-lo. Ao instalarmos o Android Studio, automaticamente também foi instalado um emulador e criado um AVD (*Android Virtual Device*) para ser usado durante o desenvolvimento. A figura a seguir demonstra a seleção do AVD. Você pode marcar a opção de lembrar qual dispositivo foi escolhido, dessa forma essa janela não será mais exibida quando você rodar o aplicativo novamente.
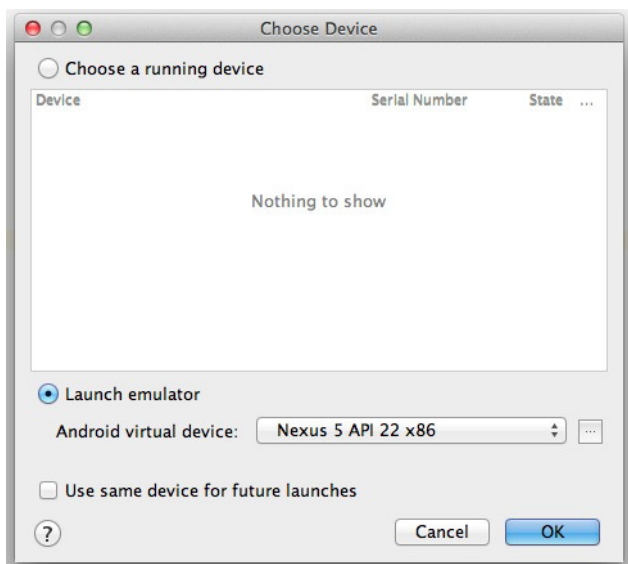
Figura 1.10: Iniciando o AVD

A inicialização do emulador pode demorar um pouco, aproveite para buscar uma xícara de café. Quando retornar, o aplicativo já deve ter sido iniciado e uma tela semelhante a esta será mostrada:
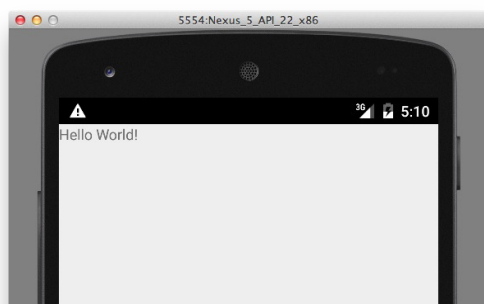
Figura 1.11: Hello Android!

Dica

Não é necessário reiniciar o emulador para testar uma nova versão. A dica é mantê-lo sempre aberto para economizar o tempo de inicialização.

1.3 CONHEÇA A ESTRUTURA DO PROJETO

Vamos conhecer agora a estrutura básica de um projeto. Para facilitar, a organização o Android Studio trabalha com o conceito de visões para agrupar os arquivos em pastas, seguindo uma determinada lógica e com determinado objetivo. A visão que utilizamos até agora foi a *Android*, que organiza os arquivos para simplificar o acesso aos itens mais usuais no desenvolvimento de um aplicativo, que são: os manifestos, o código-fonte e os recursos adicionais como arquivos de layout, ícones e imagens.

Na figura seguinte, podemos identificar essas pastas principais (`manifests` , `java` , `res`), além de uma referência para os scripts do Gradle, usados para compilar e construir o aplicativo:
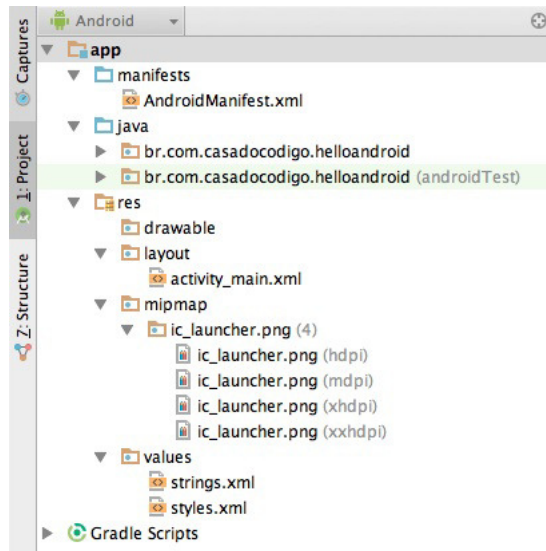
Figura 1.12: Estrutura do projeto

- `manifests` — armazena o `AndroidManifest.xml` , obrigatório para toda aplicação Android. Esse arquivo contém informações essenciais sobre a sua aplicação e sobre o que é necessário para executá-la. O nome do pacote escolhido durante a criação do projeto, por exemplo, é armazenado lá para servir como identificador único da sua aplicação.

- `java` — pasta dedicada aos armazenamentos dos códigos-fontes do projeto e será onde colocaremos as classes Java que criaremos em nossa aplicação.

- `res` — dedicado ao armazenamento de recursos (arquivos de layout, imagens, animações e `XML` contendo valores como `strings` , `arrays` etc.), acessíveis por meio da classe `R` .

Em grande parte dos projetos também é comum termos mais duas pastas para armazenar itens específicos:

- `assets` — diretório para o armazenamento de arquivos diversos utilizados por sua aplicação. Diferentemente dos recursos armazenados na pasta `res`, estes são acessíveis apenas programaticamente.
- `libs` — pasta para armazenar bibliotecas de terceiros que serão utilizadas pela aplicação e que não estão disponíveis por meio de um repositório compatível.

Na maior parte do tempo, você usará esse tipo de visão no Android Studio. Clicando no nome da visão selecionada, é possível alternar entre as visões disponíveis. Outra que também é bastante utilizada é a *Project*. Esta permite visualizar todos os arquivos existentes no projeto da mesma forma em que eles se encontram armazenados no seu computador.

> **DICA: ACESSANDO O AVD MANAGER E SDK MANAGER PELO ANDROID STUDIO**
>
> Pelo menu `Tools > Android`, é possível fazer um acesso rápido ao AVD Manager e SDK Manager para baixar novas versões do Android e também criar novos dispositivos virtuais!

1.4 HELLO WORLD 2.0

Para melhorar a nossa aplicação, vamos incluir mais algumas coisas e aproveitar para entender alguns pontos fundamentais do desenvolvimento Android. Em nossa versão melhorada do Hello World, o usuário informará seu nome em uma caixa de texto, pressionará um botão e a aplicação apresentará uma saudação personalizada. A aplicação ficará com a seguinte aparência:
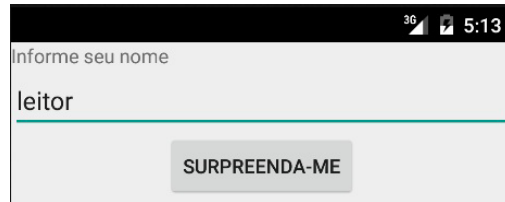
Figura 1.13: Versão melhorada

Podemos modificar o layout `activity_main.xml` já existente para incluir um campo onde o usuário deve informar o seu nome. Esse campo pode ser criado utilizando um *widget* do tipo `EditText`, no qual podemos, inclusive, indicar que ele receberá o foco da aplicação:

```
<EditText
    android:id="@+id/nomeEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="textPersonName" >

    <requestFocus />
</EditText>
```

Além disso, também teremos que incluir um botão, por meio do widget `Button`:

```
<Button
    android:id="@+id/saudacaoButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="@string/surpreenda_me" />
```

Com estes novos elementos, podemos montar a tela completa do novo *Hello World*, que ficará como o código a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

```xml
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/titulo" />

    <EditText
        android:id="@+id/nomeEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPersonName" >

        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/saudacaoButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/surpreenda_me" />

    <TextView
        android:id="@+id/saudacaoTextView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Nos itens recém-adicionados, repare que colocamos um atributo `android:id` , que é importante, pois posteriormente precisaremos referenciar e manipular esses componentes visuais. Também é necessário criar as `strings` que serão utilizadas como o título, o rótulo do botão e também uma saudação. Nosso arquivo `strings.xml` deverá ficar assim:

```xml
<resources>
    <string name="app_name">HelloAndroid</string>
    <string name="hello_world">Hello Android!</string>
    <string name="titulo">Informe seu nome</string>
    <string name="surpreenda_me">Surpreenda-me</string>
    <string name="saudacao">Olá</string>
</resources>
```

Ao executarmos a aplicação novamente, já perceberemos as

mudanças realizadas e o resultado será semelhante à figura anterior. Como ainda não programamos nenhuma ação para o botão disponível na tela, ao pressioná-lo, nada de diferente acontece.

Para obter o resultado esperado, criaremos um método na nossa `MainActivity` que responderá a esta ação apresentando ao usuário uma saudação personalizada. Logo, vamos configurar nosso botão para que, quando ele for pressionado, um método seja invocado. Para isso, utilizaremos o `onClick` :

```
<Button
        android:id="@+id/saudacaoButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/surpreenda_me"
        android:onClick="surpreenderUsuario"/>
```

Informamos que o método a ser acionado após o clique do botão é o `surpreenderUsuario` , por meio da propriedade `onClick` . Este **necessariamente** deve ser público e receber como parâmetro um objeto do tipo `View` , que é uma referência ao botão que foi pressionado:

```
public void surpreenderUsuario(View v){

}
```

Nesse método, precisamos modificar o conteúdo do widget `saudacaoTextView` , para que ele mostre o conteúdo informado no `EditText` . Para isso, a classe `MainActivity` deve possuir referência para esses elementos. Vamos começar declarando atributos para o `EditText` e o `TextView` :

```
    public class MainActivity extends Activity {
        private EditText nomeEditText;
        private TextView saudacaoTextView;

        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.main);
    }

    public void surpreenderUsuario(View v) { }

}
```

No método `onCreate`, temos que conseguir as referências para os componentes. Podemos fazer isso por meio do método `findViewById`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    this.nomeEditText =
        (EditText) findViewById(R.id.nomeEditText);
       this.saudacaoTextView = (TextView)
        findViewById(R.id.saudacaoTextView);
}
```

Agora que já temos as referências para os objetos, podemos obter o valor digitado pelo usuário, que está armazenado no `EditText`, e atribuí-lo como conteúdo do `TextView`. Para realizar essa operação, basta implementar o método `surpreenderUsuario` dessa forma:

```
public void surpreenderUsuario(View v) {
    Editable texto = this.nomeEditText.getText();
    this.saudacaoTextView.setText(texto);
}
```

Por fim, vamos adicionar a `string` `saudacao`, que está definida no arquivo `strings.xml`, para compor a mensagem final para o usuário. Para isso, basta recuperá-la através do método `getResources`:

```
String saudacao = getResources().getString(R.string.saudacao);
```

Podemos fazer a leitura dessa saudação no método `onCreate` e usar a mensagem na nossa implementação de `surpreenderUsuario`, como no código a seguir:

```
public class MainActivity extends Activity {

    private EditText nomeEditText;
    private TextView saudacaoTextView;
    private String saudacao;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        this.nomeEditText =
            (EditText) findViewById(R.id.nomeEditText);
           this.saudacaoTextView =
            (TextView) findViewById(R.id.saudacaoTextView);
           this.saudacao =
            getResources().getString(R.string.saudacao);
    }

    public void surpreenderUsuario(View v) {
        Editable texto = this.nomeEditText.getText();
        String msg = saudacao + " " + texto.toString();
        this.saudacaoTextView.setText(msg);
    }
}
```

Agora podemos executar essa aplicação que será similar à figura:
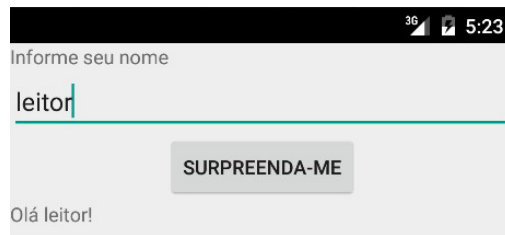
Figura 1.14: HelloWorld 2.0

1.5 CONCLUSÃO

Neste capítulo, aprendemos um pouco da história do Android e suas versões, e tivemos o primeiro contato com o Android SDK e com o Android Studio. Criamos um projeto de exemplo, buscamos entender seus detalhes e organização, além de fazer modificações no código e layout para termos nossa primeira experiência. No capítulo

seguinte, abordaremos itens essenciais para o desenvolvimento Android.

CAPÍTULO 2

ENTENDA O FUNCIONAMENTO DO ANDROID

Após desenvolver nosso primeiro projeto na plataforma Android, é importante conhecer um pouco mais dos elementos que a compõem e também do funcionamento básico do Android. Este capítulo apresenta como as aplicações são geradas e executadas, quais são os componentes de aplicação existentes na plataforma, como eles comunicam-se e também como os elementos de interface gráfica estão organizados.

2.1 A EXECUÇÃO DAS APLICAÇÕES

Até a versão 4.4 do Android, as aplicações implementadas utilizando a linguagem Java eram executadas por meio de uma máquina virtual, baseada em registradores, e otimizada para consumir pouca memória, chamada Dalvik. Ao contrário da máquina virtual Java que executa *bytecodes*, a Dalvik utiliza arquivos no formato `.dex` gerados a partir de classes Java compiladas. Esta conversão é feita pela ferramenta `dx` que acompanha o Android SDK.

Basicamente, o que é feito é o agrupamento de informações duplicadas que se encontram espalhadas em diversos arquivos

`.class` em um arquivo `.dex`, com tamanho menor do que os que o originaram. O `dx` também faz a conversão de bytecodes para um conjunto de instruções específico da máquina virtual Dalvik.

Depois de criado, o arquivo `.dex` e todos os recursos utilizados na aplicação, como imagens e ícones, são adicionados em um arquivo `.apk`, que é o aplicativo propriamente dito, capaz de ser instalado em um dispositivo. Esses arquivos encontram-se na pasta `bin` do projeto.

Já no Android 5.0, foi introduzido o novo *Android Runtime* (ART), que faz uso da compilação *ahead-of-time* (AOT), permitindo que, durante a instalação, o aplicativo seja compilado e otimizado para o dispositivo alvo. Além disso, o ART trouxe melhorias nos mecanismos de coleta de lixo, de depuração e de *profiling*. As informações necessárias para o AOT são as mesmas existentes nos arquivos `.dex`.

É possível distribuir sua aplicação para outras pessoas apenas fornecendo o arquivo `.apk`. No entanto, para colocá-la na loja Google Play, alguns outros passos são necessários, os quais serão detalhados em outro momento. A figura a seguir demonstra o processo de geração do aplicativo.
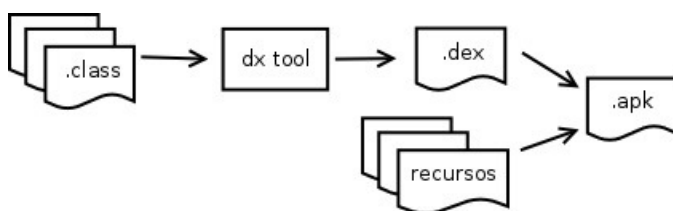
Figura 2.1: Processo de geração do aplicativo

No sistema operacional Android, para cada aplicação é atribuído um usuário único de sistema e apenas este recebe permissões para acessar seus arquivos. Além disso, por padrão, cada

aplicação é executada em um processo próprio, que possui também sua própria instância da máquina virtual Dalvik. Sendo assim, ela é executada de forma segura e isolada das demais.

Neste contexto, uma aplicação não pode acessar arquivos de outra e tampouco acessar diretamente recursos do sistema operacional — como a lista de contatos, câmera, GPS e rede —, sem que o usuário explicitamente autorize o acesso durante a instalação. Diante dessas restrições de segurança, como tiramos proveito de toda a infraestrutura do Android e também de aplicativos de terceiros para incrementar as funcionalidades da nossa aplicação, incluindo por exemplo, um recurso de capturar fotos e vídeos, e compartilhar via e-mail? Esse é justamente o ponto que vamos abordar na próxima seção.

2.2 CONHEÇA AS INTENTS E INTENT FILTERS

As `Intents` geralmente são criadas a partir de ações do usuário e representam a *intenção* de se realizar algo, como iniciar o aplicativo de correio eletrônico do Android ou iniciar a reprodução de uma música. Formalmente, as `Intents` podem ser definidas como mensagens enviadas por um componente da sua aplicação (uma `activity` , por exemplo) para o Android, informando a intenção de inicializar outro componente da mesma aplicação ou de outra.

A figura a seguir demonstra as opções apresentadas pelo Android que correspondem às aplicações capazes de tratar a intenção informada pelo usuário. Neste exemplo, desejou-se compartilhar um texto selecionado.
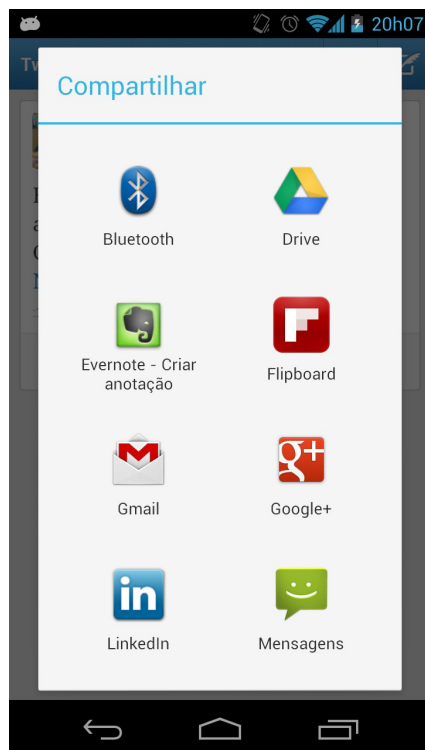
Figura 2.2: Aplicações que podem responder à intenção de compartilhar

Este é um recurso-chave no Android, pois é por meio dele que podemos fazer com que as aplicações colaborem entre si, disponibilizando funcionalidades que podem ser reutilizadas, sem a necessidade de importar códigos ou dependências para dentro da sua aplicação. Através de `Intents` , é possível iniciar novas `activities` , como fazer uma busca e selecionar um contato do telefone, abrir a aplicação de mapas com as coordenadas de localização do GPS, abrir uma página da web, tirar fotos utilizando a câmera etc., apenas reaproveitando funcionalidades já existentes, disponibilizadas pelos aplicativos instalados no aparelho.

Além disso, aplicativos de terceiros, assim como os nossos, podem disponibilizar novas funcionalidades acessíveis via

Intents . Existem, por exemplo, aplicativos de leitura de códigos de barra que podem ser chamados pela sua aplicação, utilizando a câmera do aparelho, para lê-los, e devolver o resultado para ser processado por um método da sua aplicação. Podemos criar e utilizar as Intents de diversas maneiras; a seguir veremos alguns exemplos. O trecho de código adiante mostra como abrir uma página usando o navegador que acompanha o Android:

```
Uri uri = Uri.parse("http://www.android.com");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Uma Uri foi criada a partir de uma string representando a URL que desejamos visitar. Em seguida, instanciamos uma nova Intent informando a ação que gostaríamos de executar (Intent.ACTION_VIEW), juntamente com a Uri criada, e chamamos o método startActivity da classe Activity passando a intent .

Repare que não indicamos exatamente a activity que deve ser iniciada para abrir o site desejado. Neste caso, a nossa Intent **é classificada como implícita**. Com base na ação Intent.ACTION_VIEW e o no conteúdo da sua Uri , o Android decide qual é a atividade mais adequada para resolver a URL informada. Neste caso, o escolhido é o navegador.

A seguir, temos um exemplo de como iniciar uma nova atividade existente na nossa aplicação, passando no construtor da Intent a classe correspondente à atividade que deve ser iniciada.

```
Intent intent = new Intent(this, OutraAtividade.class);
startActivity(intent);
```

Diferentemente do exemplo anterior, agora nós informamos **exatamente** qual atividade deve ser iniciada; ou seja, agora nossa Intent é **explícita**. Geralmente, as explícitas são utilizadas apenas para interação entre componentes de uma mesma aplicação, já que é

necessário conhecer o componente que deverá ser ativado; enquanto as implícitas são usadas para ativar componentes de outra aplicação, fornecendo informações adicionais, como a ação e `Uri` , para que o Android localize o adequado.

Outra característica importante é que podemos colocar informações extras na `Intent` que serão usadas posteriormente pelo componente iniciado por ela. Para exemplificar, considere que a nossa aplicação deve tirar uma foto e armazená-la em uma pasta específica.

O Android já possui um aplicativo que realiza essa tarefa, e o que queremos é chamá-lo a partir da nossa aplicação, para capturar a imagem e salvá-la em um local determinado. O código a seguir inicia a `Activity` de câmera do aparelho, informando o local e nome desejado para o armazenamento da imagem capturada:

```
/*
    O exemplo considera que existe a pasta LivroDeAndroid
    e que o aplicativo tem permissão de escrita.
*/

Uri uri = Uri.fromFile(
    new File("/sdcard/LivroDeAndroid/hello_camera.jpg"));

Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
intent.putExtra(MediaStore.EXTRA_OUTPUT, uri);

startActivity(intent);
```

Esse exemplo também utiliza `Intents` implícitas, pois em nenhum momento foi indicado qual classe de `Intent` deveria ser usada. Em resumo, uma `Intent` **é o conjunto de informações necessárias para ativar um componente de uma aplicação**. É composta basicamente de 5 informações.

Nome do componente

O **nome do componente** é definido pelo nome completo da

classe e o nome do pacote definido no `AndroidManifest.xml` que representam o componente que deve ser o encarregado de tratar a `Intent` .

Quando criamos uma `Intent` explícita com o construtor `Intent(this, OutraAtividade.class)` , o nome do componente é criado automaticamente. No entanto, também é possível defini-lo de forma programática, utilizando os métodos `setComponent()` , `setClass()` ou `setClassName()` da classe `Intent` .

Ação

A **ação** é uma `string` que define o que deve ser realizado. Existem diversas ações genéricas no Android, disponibilizadas como constantes na classe `Intent` . Alguns exemplos de constantes são:

- `ACTION_CALL` — indica que uma chamada telefônica deve ser realizada.
- `ACTION_VIEW` — indica que algum dado deve ser exibido para o usuário.
- `ACTION_EDIT` — indica que se deseja editar alguma informação.
- `ACTION_SENDTO` — indica que se deseja enviar alguma informação.

Enquanto a `Intent` declara **o que** deve ser feito, o componente que a recebe é o responsável por definir **como** a ação será executada. Ou seja, para uma mesma ação, podemos ter comportamentos distintos, quando ela for executada por diferentes componentes. Um exemplo disso é a `ACTION_VIEW` , que pode ser utilizada tanto para indicar que desejamos abrir uma página da internet quanto para abrir informações de um contato armazenado no telefone.

Dados

Os **dados** de uma `Intent` são representados por meio de uma URI e, a partir dela, a aplicação decide o que deve ser feito. No primeiro exemplo de uso de `intents`, criamos uma `Uri` para a página que gostaríamos de visitar. Outro exemplo seria criar uma `Intent` informando uma `Uri` com valor "`content://contacts/people/`", que abriria os contatos do telefone:

```
Uri uri = Uri.parse("content://contacts/people/");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Informações extras

As **informações extras** são quaisquer outros dados necessários para que o componente execute a ação apropriadamente. Elas podem ser informadas por meio dos `extras` da `Intent`.

No exemplo anterior, passamos uma `Uri` como extra para informar o local onde a foto deveria ser armazenada. Além disso, podemos também indicar outros tipos de dados, como: `strings`, tipos primitivos, `arrays` e objetos serializáveis. Para incluir um dado como extra, usamos o método `putExtra` da classe `Intent`, fornecendo uma `string` como identificador do dado e o seu respectivo valor. Relembre com o código a seguir, no qual o `MediaStore.EXTRA_OUTPUT` é o identificador da informação e a `uri` é o extra:

```
intent.putExtra(MediaStore.EXTRA_OUTPUT, uri);
```

Categoria

A **categoria**, representada apenas por uma `string`, serve como informação adicional para auxiliar o Android na escolha de qual componente é o mais adequado para receber a `Intent`. Podemos

adicionar várias categorias a uma `Intent` através do método `addCategory`. Assim como as ações, existem várias categorias predefinidas, como a `Intent.CATEGORY_APP_MUSIC`, que, quando colocada em uma `Intent`, informará ao Android que uma `Activity` capaz de reproduzir músicas deve ser acionada.

2.3 COMO AS INTENTS SÃO RESOLVIDAS

As informações contidas nas `Intents` são utilizadas pelo Android para localizar o componente adequado, geralmente uma `activity`, para executar a ação desejada. Quando o nome de componente é informado, o Android inicializa exatamente aquele componente, sem necessidade de avaliar a ação ou categoria.

Por outro lado, quando o nome não é informado, é necessário consultar quais são os componentes existentes com a habilidade de executar a ação desejada e pertencentes às categorias existentes na `Intent`. Adicionalmente, o Android também pode procurar por componentes capazes de resolver a `Uri` repassada e também de lidar com o formato dos dados, o *MIME type*, informado.

A pergunta que deve estar latente é: como o Android sabe ou encontra a `Activity` que deve ser iniciada, apenas informando esses dados na `Intent`? A resposta é que não existe mágica; em algum lugar deve estar especificado que determinadas ações podem ser resolvidas por um dado componente.

A definição de quais ações um componente está apto a responder, bem como a quais categorias ele pertence e quais dados ele sabe tratar, é realizada por meio de `intent filters`, que são configurados no arquivo `AndroidManifest.xml`.

No nosso primeiro exemplo, já existe a declaração de um `intent filter` no `AndroidManifest.xml` para a `Activity`

principal da nossa aplicação:

```
...
<activity
    android:name=".MainActivity"
    android:label="@string/title_activity_main" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
            android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
...
```

Este `intent filter` indica que a atividade `MainActivity` é aquela que deve ser iniciada ao abrir a aplicação e que também deve ser listada como uma aplicação do Android que pode ser usada por um usuário. Os `intent filters` podem ainda declarar, além da ação e da categoria, os tipos de dados com os quais o componente é capaz de lidar, como uma imagem, por exemplo:

```
<data android:mimeType="image/*" />
```

Com base nestas três informações (`action` , `category` e `data`), o Android é capaz de selecionar qual é o componente mais adequado para responder a uma `Intent` implícita, comparando o que foi passado na `Intent` com aquilo que está declarado nos `intent filters` dos aplicativos. Nossas aplicações podem definir `intent filters` com ações e categorias próprias ou fazer uso das já existentes para expor funcionalidades para as demais aplicações.

2.4 CONSTRUÇÃO DA NOSSA PRIMEIRA INTENT

Agora que já sabemos utilizar as `Intents` e compreendemos os `intent filters` , vamos tirar proveito disto, alterando o nosso HelloWorld 2.0 para incluir uma `activity` que vai responder a uma `intent` implícita.

A ideia é que na nossa aplicação de exemplo existam duas atividades: a `MainActivity` , que continuará sendo utilizada para o usuário informar o seu nome; e a `SaudacaoActivity` , que será responsável apenas por exibir uma saudação para o usuário a partir das informações contidas na `intent` . Ela também possuirá uma categoria própria e responderá a uma ação específica. A figura a seguir demonstra a ideia.
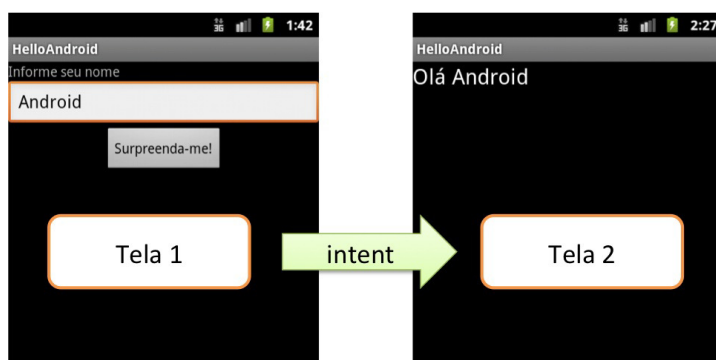
Figura 2.3: HelloWorld utilizando intents

Primeiramente, devemos criar um novo `XML` de layout que será usado pela `SaudacaoActivity` para exibir a mensagem de saudação para o usuário. Para isto, acesse o menu `File > New > Android XML File` . Informe o nome do arquivo como `saudacao` , não é necessário alterar nenhuma outra informação. Neste novo layout, incluiremos apenas um `TextView` para mostrar a saudação ao usuário. O código ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/saudacaoTextView"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Com o layout pronto, vamos criar uma nova `Activity` para a aplicação, pelo menu `File > New > Java Class`. Lembre-se de que essa opção só estará disponível se você estiver com o nome do pacote selecionado na visão Android da IDE, pois as opções são contextuais. Na caixa de diálogo apresentada, para o nome da classe informe `SaudacaoActivity` e finalize. Podemos criar a nossa `SaudacaoActivity`, que herdará de `Activity`:

```
public class SaudacaoActivity extends Activity {

    // teremos que implementar o método onCreate

}
```

Na implementação do método `onCreate`, teremos a chamada para `super.onCreate` e, em seguida, precisamos indicar qual o `layout` será utilizado, que, no nosso caso, será o `saudacao`:

```
public class SaudacaoActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.saudacao);
    }
}
```

Podemos definir constantes para identificar o extra que a `intent` possui, pois vamos usá-la nos métodos `onCreate`:

```
public class SaudacaoActivity extends Activity {
    public static final String EXTRA_NOME_USUARIO =
                        "helloandroid.EXTRA_NOME_USUARIO";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.saudacao);
    }
}
```

Com isso, podemos recuperar a `Intent`, que nos foi passada através do método `getIntent`, e checar se existe um extra com o identificador definido; ou seja, se ela possui o nome do usuário para a exibição da saudação. Caso exista um extra, obtemos o seu valor utilizando o método `intent.getStringExtra(EXTRA_NOME_USUARIO)`. Se a `intent` fornecida não possui nenhum, apresentamos um aviso para o usuário.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.saudacao);

    TextView saudacaoTextView =
        (TextView) findViewById(R.id.saudacaoTextView);

    Intent intent = getIntent();
    if (intent.hasExtra(EXTRA_NOME_USUARIO)) {
        String saudacao =
            getResources().getString(R.string.saudacao);
        saudacaoTextView.setText(saudacao + " " +
            intent.getStringExtra(EXTRA_NOME_USUARIO));
    } else {
        saudacaoTextView.setText("O nome do usuário " +
            " não foi informado");
    }
}
```

Em seguida, precisaremos alterar o método `surpreenderUsuario` da `MainActivity` para deixar de exibir a saudação e criar a `Intent` que acionará a nova atividade. Com isso, teremos que definir também a ação e a categoria da que acabamos de criar. Podemos fazer isso definindo mais duas constantes na classe `SaudacaoActivity`:

```
public class SaudacaoActivity extends Activity {
    public static final String EXTRA_NOME_USUARIO =
        "helloandroid.EXTRA_NOME_USUARIO";

    // As duas novas constantes
    public static final String ACAO_EXIBIR_SAUDACAO =
        "helloandroid.ACAO_EXIBIR_SAUDACAO";
```

```
    public static final String CATEGORIA_SAUDACAO =
        "helloandroid.CATEGORIA_SAUDACAO";

    // método onCreate
}
```

No método `surpreenderUsuario`, criamos uma nova `Intent` com a ação desejada e nela adicionamos a categoria definida anteriormente. Em seguida, incluímos como informação extra o valor informado no `EditText`. Por fim, iniciamos uma nova `activity`, passando a `Intent` criada.

```
public void surpreenderUsuario(View v) {
    Intent intent =
        new Intent(SaudacaoActivity.ACAO_EXIBIR_SAUDACAO);

    intent.addCategory(SaudacaoActivity.CATEGORIA_SAUDACAO);

    String texto = nomeEditText.getText().toString();
    intent.putExtra(SaudacaoActivity.EXTRA_NOME_USUARIO, texto);
    startActivity(intent);
}
```

Já que a exibição da mensagem de saudação é responsabilidade de outra atividade, podemos excluir do layout, usado pela `MainActivity`, o `TextView` que tinha esse papel. No arquivo de layout `activity_main.xml`, remova o último `TextView` declarado, com o `id` `@saudacaoTextView`, pois não precisaremos mais dele.

As últimas alterações para que nossa `SaudacaoActivity` possa responder a uma `Intent` serão feitas no arquivo `AndroidManifest.xml`. Vamos adicionar um novo bloco de `activity` declarações, nas quais estabelecemos que a `SaudacaoActivity` responde pela `ACAO_EXIBIR_USUARIO` e também atende a `intents`, que pertencem a `CATEGORIA_SAUDACAO`:

```
<activity
    android:name=
```

```
        "br.com.casadocodigo.helloandroid.SaudacaoActivity">
    <intent-filter>
        <action
            android:name="helloandroid.ACAO_EXIBIR_SAUDACAO" />
        <category
            android:name="helloandroid.CATEGORIA_SAUDACAO" />
        <category
            android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Quando desejamos que uma `activity` receba `intents` implícitas, é obrigatório que no `intent filter` também seja incluída a categoria `android.intent.category.DEFAULT`. Para cada `activity`, é possível definir vários `intent filters`, com configurações diferentes de ação e categoria. Já podemos executar a aplicação para testar! O resultado deve ser o mesmo apresentado na figura *HelloWorld utilizando intents*.

2.5 COMPONENTES DE APLICAÇÃO

Até aqui já desenvolvemos uma aplicação de exemplo e já tivemos contato com um dos componentes mais importantes da plataforma Android, que são as `Activities`. Agora chegou o momento de conhecer quais são os outros tipos de componentes.

Para construir uma aplicação Android, podemos utilizar quatro tipos de componentes, cada qual com um propósito e ciclo de vida bem definidos, são eles:

- **Activities** — uma atividade representa uma tela com interface gráfica capaz de promover algum tipo de interação com o usuário. Já utilizamos este tipo de componente para implementar nossa primeira aplicação. Uma aplicação Android pode ser composta de diversas *activities* para fornecer um conjunto de funcionalidades para o usuário.

- **Services —** os serviços são componentes executados em segundo plano e que não dispõem de interface gráfica. Seu objetivo principal é realizar tarefas que podem consumir muito tempo para executar, sem comprometer a interação do usuário com alguma `activity` . Tocar uma música ou fazer o download de um arquivo são exemplos de funcionalidades que podem ser implementadas usando *services*.

- **Content providers —** os provedores de conteúdo são componentes que permitem o acesso e a modificação de dados armazenados em um banco de dados SQLite local, de arquivos armazenados no próprio dispositivo ou mesmo dados armazenados na web. Os *content providers* podem ser expostos para uso por outras aplicações, com o objetivo de compartilhar dados, ou podem ser utilizados apenas pela aplicação que os contém.

- **Broadcast receivers —** são componentes capazes de responder a eventos propagados pelo sistema operacional Android, por exemplo o nível baixo da bateria, ou eventos originados por uma aplicação, como o recebimento de uma nova mensagem de texto.

Não é necessário que uma aplicação Android tenha todos esses componentes, mas é importante conhecê-los para que, no momento de projetar a aplicação, possamos selecionar o adequado para atender às necessidades. Nos capítulos *Compartilhe dados entre aplicações com os content providers* e *Integração de aplicações Android com serviços REST*, vamos explorar mais o uso de cada componente.

2.6 CICLO DE VIDA DA ACTIVITY

A `Activity` é um componente de aplicação com um ciclo de vida específico. Quando o usuário acessa a aplicação, navega pelas opções, sai ou retorna a ela, as atividades que a compõem passam por uma série de estados do ciclo de vida. Entender como ele funciona é importante para preparar a aplicação para lidar com situações que podem interferir na sua execução, tais como o recebimento de uma ligação, desligamento da tela do aparelho, ou ainda a abertura de outra aplicação feita pelo usuário. A figura ilustra seu ciclo de vida:
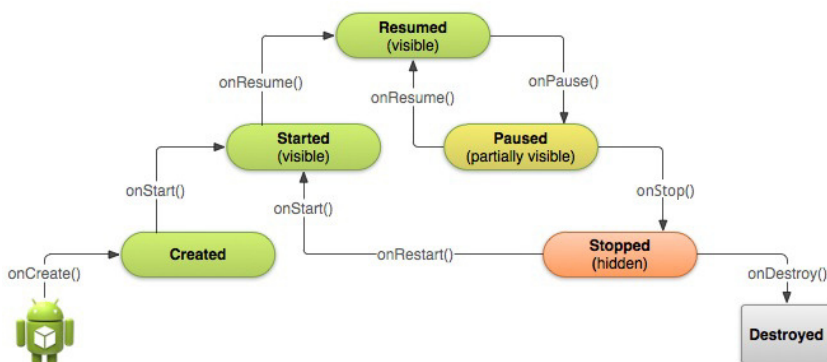
Figura 2.4: Ciclo de vida da Activity. Fonte: developer.android.com

Sempre que a `Activity` muda de estado, o Android aciona um método (`callback`) correspondente. Assim que o usuário inicia uma aplicação, o Android cria a atividade principal que está declarada no `AndroidManisfest.xml` e invoca o seu método `onCreate` . Como já vimos, é neste que atribuímos qual layout será utilizado pela nossa atividade e também inicializamos variáveis e recursos necessários.

Em seguida, o Android invoca os métodos `onStart` e, logo após, o `onResume` . A `Activity` torna-se visível para o usuário no estado `Started` e assim permanece até os métodos `onPause`

(visível parcialmente) ou `onDestroy` serem chamados. Quando ela está no estado `Resumed` , dizemos que ela está no *foreground* e pode realizar interação com o usuário.

Ela muda para o estado `Paused` quando for parcialmente encoberta por outra `Activity` , que pode não ocupar toda a tela ou ser transparente. Se o usuário sair da aplicação ou iniciar outra atividade que encubra totalmente a que está sendo executada, o método `onStop` é invocado e a `Activity` vai para o `background` . Mesmo não sendo mais visível pelo usuário, ela continua instanciada e com seu estado interno inalterado; ou seja, da forma como estava quando em execução.

Quando uma `Activity` está nos estados de `Paused` ou `Stopped` , o sistema operacional pode removê-la da memória, invocando o seu método `finish` ou encerrando arbitrariamente o seu processo. Nestas condições, o método `onDestroy` é disparado. Após destruída, se ela for aberta novamente, será recriada.

Podemos sobrescrever esses métodos para acrescentar ações que devem ser realizadas em determinado estágio do ciclo de vida. Por exemplo, quando a `Activity` não estiver mais visível, podemos liberar recursos, como uma conexão de rede, ou ainda, salvar os dados digitados pelo usuário no método `onPause` e encerrar as threads em execução no método `onDestroy` . O código a seguir mostra os métodos que podemos sobrescrever:

```
public class MinhaActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // A activity está sendo criada
    }
    @Override
    protected void onStart() {
        super.onStart();
        // A activity está prestes a se tornar visível
    }
```

```
    @Override
    protected void onResume() {
        super.onResume();
        // A activity está visível
    }
    @Override
    protected void onPause() {
        super.onPause();
        /* Outra activity está recebendo o foco. Esta activity
         ficará pausada */
    }
    @Override
    protected void onStop() {
        super.onStop();
        // A activity não está mais visível
        // mas permanece em memória
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // A activity está prestes a ser destruída
        // (removida da memória)
    }
}
```

Lembre-se sempre de invocar a implementação padrão do método que está sendo sobrescrito. Por exemplo, se estiver sobrescrevendo o `onStop` , então invoque antes o método `super.onStop()` .

2.7 LAYOUTS, WIDGETS E TEMAS

Sem dúvida uma interface gráfica com boa usabilidade e que provê uma excelente experiência de uso, assim como funcionalidades bem implementadas, é importante para o sucesso de uma aplicação mobile. A plataforma Android nos oferece um bom conjunto de componentes visuais, os chamados *widgets*, bem como opções de layout variadas para a criação da interface com o usuário.

O elemento fundamental de uma interface gráfica na plataforma

Android é a `View`. A partir dela é que são derivados todos os demais elementos, como: botões; imagens; *checkboxes*; campos para entrada e exibição de textos; e também widgets mais complexos, como seletores de data, barras de progresso e de pesquisa, e até mesmo um para exibir páginas web, o `WebView`. A figura a seguir mostra alguns deles:
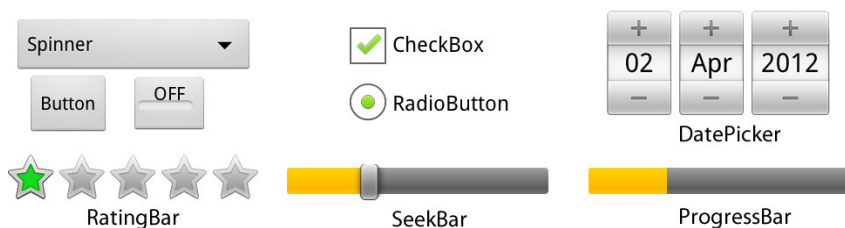
Figura 2.5: Alguns widgets disponíveis

Outra classe essencial é a `ViewGroup`, que tem como características especiais a possibilidade de conter outras `Views` e ser a base para todas as classes que constituem layouts. O diagrama da figura a seguir mostra a hierarquia desses elementos.
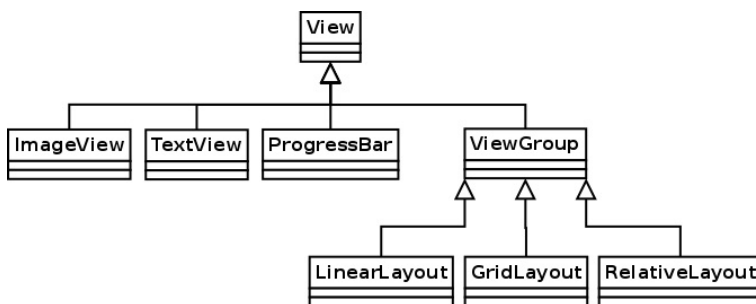
Figura 2.6: Hierarquia de Views

Outro recurso interessante disponibilizado pelo Android é a criação de estilos e temas para personalizar a sua aplicação. Se você já trabalhou com folhas de estilo CSS e design para web perceberá a similaridade entre eles. Para definir um estilo, basta criar um `XML` em `res/values/` definindo as propriedades desejadas, como no

exemplo adiante que define a cor do texto e o tipo de fonte:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="TitleFont"
            parent="@android:style/TextAppearance.Large">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#FFBA00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Há ainda a possibilidade de derivar estilos existentes. Na linha 3, fazemos isto informando qual é o estilo-pai. Para aplicar o estilo em um `TextView`, por exemplo, basta referenciá-lo dessa maneira:

```
<TextView style="@style/TitleFont"
          android:text="@string/titulo" />
```

Como é de se imaginar, os temas são conjuntos de estilos que podem ser aplicados em uma ou em todas as *activities* da aplicação. Para experimentar esse recurso, no arquivo `AndroidManifest.xml` da nossa aplicação de exemplo, podemos incluir o atributo `android:theme="@android:style/Theme.Black"` na tag `application` e executá-la novamente. Agora ela está com uma aparência mais escura como mostrado na figura:
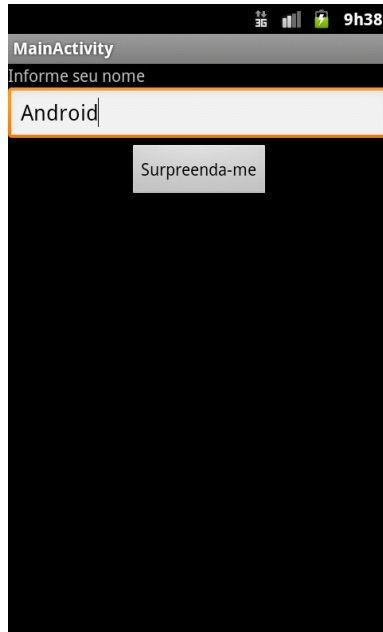
Figura 2.7: Aplicação HelloAndroid com outro tema

2.8 CONCLUSÃO

Neste capítulo, compreendemos como as aplicações são geradas e empacotadas para execução na máquina virtual Dalvik, além de uma visão geral dos componentes de aplicação (*activities*, *services*, *content providers* e *broadcast receivers*). Por meio de exemplos, exploramos um recurso-chave na plataforma Android que são as `Intents` . Além disso, foi apresentado o ciclo de vida da `Activity` e em quais situações ela muda de estado.

A organização dos elementos de interface gráfica, widgets e layouts foi apresentada, assim como a utilização de estilos e temas para a modificação da aparência da aplicação. Todos os itens mostrados aqui servem como alicerce para os demais capítulos, que fazem uso desses conceitos fundamentais para o desenvolvimento de aplicações concretas e cada vez mais elaboradas.

Capítulo 3

DOMINE OS PRINCIPAIS ELEMENTOS DE INTERFACE GRÁFICA

Neste capítulo, exploraremos a construção de interfaces gráficas e daremos início à implementação de uma aplicação que nos acompanhará durante todo o restante do livro. A aplicação que vamos desenvolver servirá para nos ajudar a controlar os gastos realizados em nossas viagens de lazer ou negócios. Não é uma ideia revolucionária, mas nos ajudará muito, pois uma aplicação assim possui um domínio muito rico, além de revelar problemas comuns do desenvolvimento para Android.

Nossa aplicação será batizada de **BoaViagem** e nela poderemos criar uma nova viagem, informando os destinos, datas de chegada e partida, e se ela é de negócios ou lazer. Para cada viagem, poderemos informar os gastos realizados por categoria, como alimentação, passeios, locomoção e hospedagem.

CÓDIGO FONTE COMPLETO DO PROJETO

Caso queira, você pode consultar o código fonte completo do **BoaViagem**, que deixei disponível no meu GitHub: https://github.com/joaobmonteiro/livro-android.

Os exemplos estão organizados por capítulo. Portanto, fique à vontade para baixar, consultar, sugerir melhorias e incrementar a aplicação.

Também se deverá informar qual o orçamento disponível para a realização da viagem. Esta informação poderá ser usada para que a aplicação nos alerte quando estivermos próximos de ultrapassar o limite de gastos estabelecido.

Estas são as funcionalidades principais e, no decorrer do livro, aprenderemos outras coisas como captura de fotos, uso do GPS e mapas, integração e compartilhamento de dados que você pode posteriormente incluir como nova funcionalidade do aplicativo. A figura a seguir mostra algumas de suas telas.
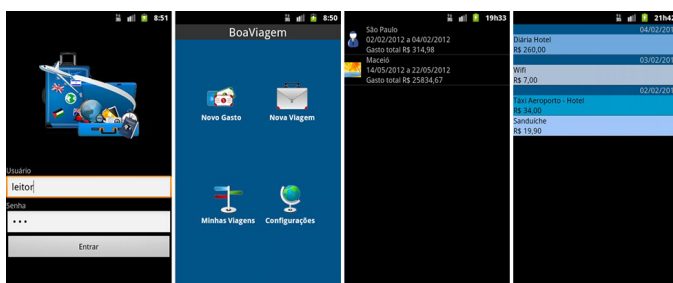
Figura 3.1: Telas do BoaViagem

Para começar, vamos criar as telas com o objetivo de conhecer os diversos tipos de layout e os widgets básicos para compor

formulários de entrada de dados. No capítulo *Persistência de dados no Android com SQLite*, incluiremos a persistência desses dados.

Então, vamos lá! Caso queira seguir codificando, crie um novo projeto Android com o nome **BoaViagem** e o pacote `br.com.casadocodigo.boaviagem` , da mesma forma que fizemos no capítulo *Construa sua primeira aplicação*.

Neste primeiro momento, nos preocuparemos com as duas telas iniciais da aplicação: a tela de login e a tela inicial de opções, comumente chama de *dashboard*. Na sequência, temos o protótipo com os detalhes dos tipos de layout que utilizaremos.

Figura 3.2: Tela de login e dashboard

3.1 LINEARLAYOUT

Para a tela de login, vamos usar o `LinearLayout` , que permite a organização dos elementos de forma linear, posicionando itens um abaixo do outro, quando configurado com orientação vertical;

ou um ao lado do outro, quando configurado na horizontal. Às vezes, escolher a orientação certa causa um pouco de confusão, então a dica é lembrar-se de que a orientação diz respeito à direção na qual os itens serão incluídos na tela. Ou seja, na orientação vertical, eles serão incluídos no layout de cima para baixo; e na horizontal, da esquerda para a direita.

Para fazer uma tela de login parecida com a do protótipo visto na figura anterior, usaremos um `LinearLayout` com orientação vertical. Para essa tela, podemos criar um novo arquivo de layout com o nome de `login.xml` . O primeiro passo é definir o `LinearLayout` que queremos e dizer que sua orientação (vertical):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_vertical"
    android:orientation="vertical" >

    <!-- Campos virão aqui -->

</LinearLayout>
```

Para especificar um layout, existem dois atributos fundamentais: o `layout_width` (linha 3), que indica a largura do elemento; e `layout_height` (linha 4), que indica a sua altura. Há dois valores importantes para estes atributos, que são o `match_parent` e o `wrap_content` . O primeiro valor informa que o tamanho deve ser o mesmo que o do elemento-pai, enquanto o segundo, que o tamanho deve ser grande o suficiente para abrigar o conteúdo a ser exibido.

Note que na linha 5, utilizamos um outro atributo, que é o `android:gravity="center_vertical"` . Ele indica que o layout deve ficar centralizado verticalmente na tela. A orientação que desejamos é informada no atributo `android:orientation` .

O próximo passo é exibir o logotipo da aplicação. Para isso, podemos incluir um `ImageView` , que deve ficar ao centro e mostrar a imagem de `android:src="@drawable/logo"` . Com isso, basta ter uma imagem em um arquivo `logo.png` e colocá-la nos diretórios `drawable` do projeto.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ...>

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginBottom="50dp"
        android:src="@drawable/logo" />

</LinearLayout>
```

Agora, podemos incluir os campos para que o usuário forneça seu login e senha para entrar na aplicação. Precisaremos de componentes `TextView` para mostrar sua descrição, como se fossem *labels*, e também do `EditText` para o campo onde o usuário digitará seu login:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ...>

    <!-- Logo -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/usuario" />

    <EditText
        android:id="@+id/usuario"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text" >

        <requestFocus />
    </EditText>
</LinearLayout>
```

Repare que no `EditText` definimos um atributo `android:inputType` , para indicar que esse campo é uma simples entrada de texto, ou seja, caracteres alfanuméricos. Também utilizamos o `<requestFocus>` , para que este receba o foco quando a tela for exibida.

Com isso, para fazermos o campo de senha, basta criarmos um novo `EditText` , cujo `android:inputType` seja do tipo `textPassword` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ...>

    <!-- Logo -->

    <!-- Campo de login do usuário -->

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/senha" />
    <EditText
        android:id="@+id/senha"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword" />

</LinearLayout>
```

COMO ADICIONAR ROLAGEM VERTICAL

Os layouts do Android não suportam por padrão a rolagem vertical da tela. Para ter essa funcionalidade, é necessário utilizar uma `ScrollView` e colocar o layout que precisa da rolagem como seu elemento-filho.

Além dos tipos `text` e `textPassword` , também existem diversos outros, tais como o `number` , que indica números e faz

com que o teclado exibido para digitação seja apenas o numérico; `phone`, para números de telefone; e `date` e `time`, para informações de data e hora. Para ver todas as opções disponíveis, consulte a documentação em http://developer.android.com/reference.

Agora, basta fazer o botão entrar, usando a Tag `Button`:

```
<Button
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom"
      android:onClick="entrarOnClick"
      android:text="@string/entrar" />
```

Nesse botão, vinculamos a ação `entrarOnClick` que fará a autenticação do usuário e, caso seja bem-sucedida, iniciará a `activity` da dashboard. Vamos implementar este método na `BoaViagemActivity`, que utilizará o layout da tela de login que acabamos de definir.

```
public class BoaViagemActivity extends Activity {

    public void entrarOnClick(View v) { }
}
```

Note que fizemos o método `entrarOnClick`, recebendo como parâmetro um objeto do tipo `View`. Vamos implementar a lógica do login dentro dele em instantes. Mas, primeiro, faremos com que, quando a `BoaViagemActivity` for criada, ela fique associada ao layout do login. Para isso, vamos reescrever o método `onCreate` e invocar o `setContentView`, passando uma referência ao layout de login.

```
public class BoaViagemActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);
    }
```

```java
    public void entrarOnClick(View v) { }
}
```

> **CLASSE R**
>
> A classe `R` é gerada automaticamente pelo Android Studio, utilizando as ferramentas do SDK, e serve para mapear os recursos existentes no aplicativo na forma de constantes. Dessa forma, podemos referenciar facilmente arquivos de layout, widgets e outros tipos de recursos como `strings` e `arrays`.

Nesse começo, faremos uma autenticação simples, na qual o usuário informado deve ser "leitor" e a senha deve ser "123". Porém, para isso, temos que recuperar uma referência para os campos de usuário e senha que estão na tela, para termos acesso aos textos presentes nos campos. No próprio método `onCreate`, vamos recuperar referências para os dois `EditText`, por meio do método `findViewById`.

```java
public class BoaViagemActivity extends Activity {
    private EditText usuario;
    private EditText senha;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);

        usuario = (EditText) findViewById(R.id.usuario);
        senha = (EditText) findViewById(R.id.senha);
    }
}
```

Note que foi passado para o `findViewById` uma referência para `R.id.usuario` e, em seguida, para `R.id.senha`. Essas são referências para os `EditText` que criamos anteriormente na tela e

demos o nome de `usuario` e `senha`.

Com esses componentes recuperados, agora podemos descobrir qual é o texto que está neles quando o método `entrarOnClick` for invocado:

```
    public void entrarOnClick(View v) {
        String usuarioInformado = usuario.getText().toString();
        String senhaInformada = senha.getText().toString();

        if("leitor".equals(usuarioInformado) &&
                "123".equals(senhaInformada)) {
            // vai para outra activity
        } else{
            // mostra uma mensagem de erro
        }
    }
}
```

No caso de falha na autenticação, deve-se exibir uma mensagem para o usuário, através do widget `Toast`, que serve para apresentar uma notificação rápida para o usuário, informando o resultado de alguma operação. É possível definir por quanto tempo a mensagem ficará visível, por meio de uma duração `Toast.LENGTH_SHORT` ou `Toast.LENGTH_LONG`.

```
if("leitor".equals(usuarioInformado) &&
        "123".equals(senhaInformada)) {
    // vai para outra activity
} else {
    String mensagemErro = getString(R.string.erro_autenticao);
    Toast toast = Toast.makeText(this, mensagemErro,
                        Toast.LENGTH_SHORT);
    toast.show();
}
```

Repare que recuperamos a mensagem de erro em um arquivo `strings.xml`, através de `R.string.erro_autenticacao`; ou seja, temos nesse arquivo uma mensagem, cujo nome é `erro_autenticacao`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <!-- mensagens padrão -->

    <string name="erro_autenticacao">
        Usuário ou senha inválidos
    </string>
</resources>
```

A única coisa que falta agora é redirecionar o usuário para outra *activity*, caso a autenticação seja feita corretamente. Para isso, criaremos uma `DashboardActivity` que vamos implementar nas próximas seções e falar que, ao autenticar, essa *activity* será iniciada:

```
if("leitor".equals(usuarioInformado)  &&
        "123".equals(senhaInformada)){
    startActivity(new Intent(this,DashboardActivity.class));
} else {
    String mensagemErro = getString(R.string.erro_autenticao);
    Toast toast = Toast.makeText(this, mensagemErro,
                        Toast.LENGTH_SHORT);
    toast.show();
}
```

3.2 RELATIVELAYOUT

O próximo passo é criar o layout e a atividade para exibir a tela de opções da nossa aplicação, o dashboard. Para esta tela, utilizaremos uma combinação do `LinearLayout`, que acabamos de ver, com o `RelativeLayout`. É bastante comum e, muitas vezes, necessário aninhar diversos tipos de layout.

O `RelativeLayout`, um dos mais poderosos e versáteis disponíveis na plataforma Android, permite posicionar um elemento em um local relativo a outro componente. É possível, por exemplo, posicionar uma imagem **abaixo** de um botão, que fica à **esquerda** de um `TextView`. Vamos criar um novo arquivo XML de layout chamado `dashboard.xml`. O objetivo é que tenhamos, além da tela de login, uma nova com as ações que podemos fazer na aplicação, como na figura:
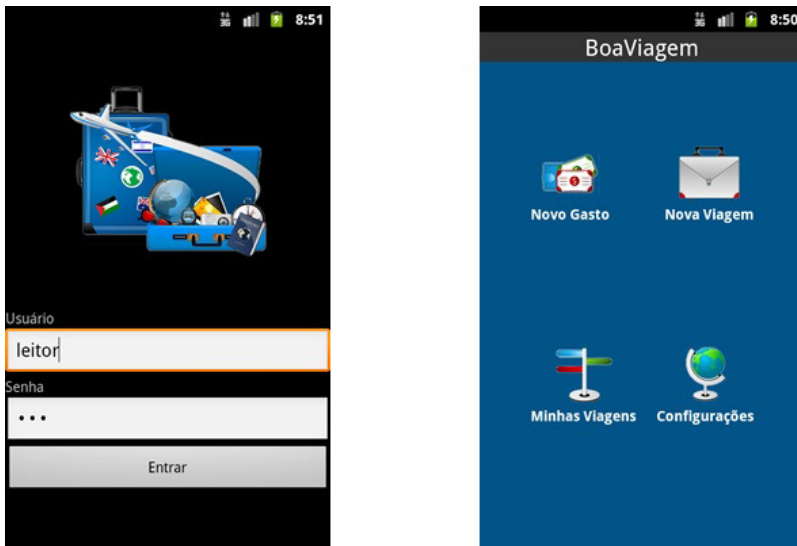
Figura 3.3: Tela de login e dashboard da aplicação

Para construirmos essa nova tela, vamos adicionar o `RelativeLayout` que deve ocupar todo o espaço restante deixado pelo `LinearLayout` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#333333"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/app_name"
        android:textAppearance=
            "?android:attr/textAppearanceLarge" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#015488">
```

```
        <!-- Vamos adicionar elementos aqui -->

    </RelativeLayout>

</LinearLayout>
```

Para representar cada opção da dashboard, usaremos `TextViews` que serão alinhados de forma relativa aos demais elementos na tela. Para posicionar seu primeiro elemento no canto superior esquerdo, incluiremos um `TextView` que define um tamanho de margem esquerda (`layout_marginLeft`) e para o topo (`layout_marginTop`).

Esse elemento não possui nenhuma informação específica de layout relativo, apenas de margens, mas servirá de referência para o próximo `TextView` . Este, por sua vez, ficará no canto superior direito. Logo, vamos colocá-lo ao lado direito do seu componente-pai (o próximo `RelativeLayout`), utilizando o atributo `layout_alignParentRight=true` .

Também é preciso alinhar o seu topo com o do `TextView` âncora referente ao "Novo Gasto". Para isto, usa-se o atributo `layout_alignTop` , informando o `id` do elemento que será a referência, no caso é o `@id/novo_gasto` . Veja:

```
<RelativeLayout ...>

    <TextView
        android:id="@+id/novo_gasto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="50dp"
        android:layout_marginTop="80dp"
        android:clickable="true"
        android:drawableTop="@drawable/novo_gasto"
        android:onClick="selecionarOpcao"
        android:text="@string/novo_gasto"
        android:textColor="#FFFFFF"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/nova_viagem"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignTop="@id/novo_gasto"
        android:layout_marginRight="50dp"
        android:clickable="true"
        android:drawableTop="@drawable/nova_viagem"
        android:onClick="selecionarOpcao"
        android:text="@string/nova_viagem"
        android:textColor="#FFFFFF"
        android:textStyle="bold" />

</RelativeLayout>
```

Também em relação ao `TextView` do novo gasto, alinhamos o item "Minhas Viagens" à esquerda e, para mantê-lo na parte de baixo da tela, especificamos que ele deve se alinhar como a região inferior do componente-pai, pela propriedade `android:layout_alignParentBottom="true"`.

O último `TextView`, para a opção de configurações da aplicação, utiliza o `TextView` anterior para se alinhar na região inferior da tela e também à direita, com base no `TextView` da "Nova Viagem". É importante ressaltar que podemos obter o mesmo resultado (mesmo design de tela), usando outros tipos de layouts ou os mesmos, mas de forma diferente das apresentadas aqui:

```
<RelativeLayout ...>

    <!-- Novo gasto e nova viagem -->

    <TextView
        android:id="@+id/minhas_viagens"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@id/novo_gasto"
        android:layout_alignParentBottom="true"
        android:layout_marginBottom="120dp"
        android:clickable="true"
        android:drawableTop="@drawable/minhas_viagens"
        android:onClick="selecionarOpcao"
        android:text="@string/minhas_viagens"
        android:textColor="#FFFFFF"
        android:textStyle="bold" />
```

```
    <TextView
        android:id="@+id/configuracoes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@id/minhas_viagens"
        android:layout_alignRight="@id/nova_viagem"
        android:clickable="true"
        android:drawableTop="@drawable/configuracoes"
        android:onClick="selecionarOpcao"
        android:text="@string/configuracoes"
        android:textColor="#FFFFFF"
        android:textStyle="bold" />

</RelativeLayout>
```

Dessa forma, o código completo, usando os dois layouts, fica assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#333333"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="@string/app_name"
        android:textAppearance=
            "?android:attr/textAppearanceLarge" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#015488" >

        <TextView
            android:id="@+id/novo_gasto"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginLeft="50dp"
            android:layout_marginTop="80dp"
            android:clickable="true"
            android:drawableTop="@drawable/novo_gasto"
```

```
            android:onClick="selecionarOpcao"
            android:text="@string/novo_gasto"
            android:textColor="#FFFFFF"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/nova_viagem"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:layout_alignTop="@id/novo_gasto"
            android:layout_marginRight="50dp"
            android:clickable="true"
            android:drawableTop="@drawable/nova_viagem"
            android:onClick="selecionarOpcao"
            android:text="@string/nova_viagem"
            android:textColor="#FFFFFF"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/minhas_viagens"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignLeft="@id/novo_gasto"
            android:layout_alignParentBottom="true"
            android:layout_marginBottom="120dp"
            android:clickable="true"
            android:drawableTop="@drawable/minhas_viagens"
            android:onClick="selecionarOpcao"
            android:text="@string/minhas_viagens"
            android:textColor="#FFFFFF"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/configuracoes"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignBottom="@id/minhas_viagens"
            android:layout_alignRight="@id/nova_viagem"
            android:clickable="true"
            android:drawableTop="@drawable/configuracoes"
            android:onClick="selecionarOpcao"
            android:text="@string/configuracoes"
            android:textColor="#FFFFFF"
            android:textStyle="bold" />
    </RelativeLayout>

</LinearLayout>
```

Dica

Utilize a visualização gráfica do layout na IDE para ter uma prévia de como está ficando a tela.

Para representar cada opção na dashboard, usamos o widget `TextView` que, além da possibilidade de exibir textos, também pode mostrar uma imagem associada. Essas imagens podem ser posicionadas à esquerda, à direita, acima ou abaixo do texto que será exibido. No nosso caso, optamos por colocá-la acima dele, informando a imagem desejada no atributo `drawableTop` . Também especificamos que nossos `TextViews` podem ser clicados, através do atributo `clickable` , e que o texto deve estar em negrito e ser da cor branca (atributos `textStyle` e `textColor` , respectivamente).

Cuidados com muitos layouts aninhados

Utilizar muitos layouts aninhados pode trazer problemas de desempenho. Prefira desenvolver layouts que possuam poucos níveis de aninhamento. Consulte o seguinte guia para obter dicas de como melhorar o desempenho nessas situações: http://developer.android.com/training/improving-layouts/index.html.

Para darmos continuidade à implementação da tela inicial de opções, vamos usar a classe `DashboardActivity` . Essa nova atividade deve usar o layout `dashboard.xml` e responder aos métodos disparados, quando uma opção for selecionada. Por

enquanto, o código dela ficará assim:

```
public class DashboardActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.dashboard);
    }

    public void selecionarOpcao(View view) {
        /*
            com base na view que foi clicada
             iremos tomar a ação correta
        */
        TextView textView = (TextView) view;
        String opcao = "Opção: "+ textView.getText().toString();
        Toast.makeText(this, opcao, Toast.LENGTH_LONG).show();
    }
}
```

É importante lembrar que todas as atividades que forem criadas devem ser declaradas no `AndroidManifest.xml`. Além disso, queremos suprimir a barra de título padrão, o que também é feito no `AndroidManifest.xml`, configurando para nossa aplicação o tema `@android:style/Theme.NoTitleBar`. Com isso, o `AndroidManifest.xml` ficará parecido com:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.casadocodigo.boaviagem"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.NoTitleBar" >
        <activity
            android:name=".BoaViagemActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
```

```
                    android:name="android.intent.action.MAIN" />

                <category
                    android:name=
                    "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".DashboardActivity" />
    </application>

</manifest>
```

Com a tela de login e dashboard prontas, podemos executar a nossa aplicação e comparar as telas criadas com os protótipos apresentados na figura *Tela de login e dashboard*. Ficaram parecidas, não é mesmo?

> **DICA**
>
> Durante o desenvolvimento dos layouts da aplicação, é comum executarmos o aplicativo várias vezes para ver como está ficando. Para evitar que seja necessário navegar e ir selecionando opções até chegar à *activity* desejada, inclua o `intent-filter` na atividade a ser testada, para que ela seja a primeira a ser iniciada.

3.3 TABLELAYOUT

As próximas telas que implementaremos serão a criação de uma nova viagem e o registro de um novo gasto. São telas tipicamente de cadastro, que permitirão que diversos widgets para entrada de dados sejam explorados, tais como os `Spinner` , `DatePicker` e nosso já conhecido `EditText` , além de mais um tipo de layout, o `TableLayout` . O protótipo das telas é o seguinte:

Figura 3.4: Telas de cadastro de viagem e de gasto

Começaremos pela tela de cadastro de uma nova viagem. Para esse layout, usaremos uma combinação de `LinearLayout` e `TableLayout` . Como o nome sugere, o `TableLayout` permite a criação de layouts com a organização em formato de tabelas, similar ao `<table>` do HTML. O elemento `TableRow` é utilizado para representar uma linha, e seus elementos-filhos representam uma célula. Por exemplo, se uma `TableRow` possui dois elementos, então aquela linha possui duas colunas.

Ainda podemos usar qualquer outra `view` que não a `TableRow` para representar uma linha. Neste caso, a `view` utilizada representa uma célula e uma única coluna para aquela linha. Ou seja, com isso temos um comportamento parecido com o `colspan` do HTML, no qual uma célula se estende por várias colunas.

O `TableLayout` também tem outra característica importante:

todos os seus elementos-filhos não podem especificar o atributo `layout_width`, que, por padrão, já possuem o valor `match_parent`. No entanto, o atributo `layout_height` pode ser definido (o valor padrão é `wrap_content`), exceto quando o elemento-filho for uma `TableRow`, que terá sempre o valor `wrap_content`.

Isto por um lado é bom, pois evita que tenhamos que especificar inúmeros atributos de layout, mas por outro, pode limitar as situações de uso do `TableLayout` por não termos este ajuste fino. Essa característica nos levou a inserir um `LinearLayout` para incluirmos um título centralizado na tela. Ainda utilizamos uma `ScrollView` para prover a rolagem da tela quando necessário.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:stretchColumns="0,1,2" >

        <LinearLayout
            android:background="#015488"
            android:orientation="vertical" >

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center"
                android:text="@string/nova_viagem"
                android:textAppearance=
                        "?android:attr/textAppearanceLarge"
                android:textStyle="bold" />
        </LinearLayout>

</ScrollView>
```

O `TableLayout` possui dois atributos muito úteis para ajustar

o tamanho das colunas. Um deles é o stretchColumns (linha 12), que permite que as colunas indicadas ocupem todo o espaço disponível entre os seus elementos-filhos e o próprio TableLayout . Como a configuração de largura do nosso layout é match_parent , logo as colunas devem expandir o seu tamanho de modo a ocupar toda a tela.

O outro atributo é o shrinkColumns que indica que determinadas colunas podem ter seu tamanho reduzido, quando requisitado pelo TableLayout . O valor desses atributos é o índice (iniciando em 0) referente às colunas desejadas. O TableLayout adiciona as linhas e os elementos de layout na sequência em que aparecem no XML, incrementando automaticamente o índice da coluna.

Se necessário, é possível informar a qual coluna determinado elemento pertence, usando o atributo layout_column . Porém, a ordem sequencial deve ser respeitada, ou seja, não é possível incluir um elemento na coluna de índice 1 e depois um na coluna de índice 0 , nem via XML, nem programaticamente. Quando um número da sequência é omitido, o TableLayout assume que aquela é uma coluna vazia. Portanto, fazer uso do layout_column faz mais sentido quando desejamos incluir colunas vazias para criar espaço entre os elementos e possivelmente incluí-las no shrinkColumns .

Agora, basta colocarmos os elementos que irão compor o layout e cada uma das colunas. Nas duas primeiras linhas da tabela, teremos a caixa de texto para que seja informado o destino e, logo acima dela, um TextView para o label do campo:

```
<TextView android:text="@string/destino" />

<EditText
    android:id="@+id/destino"
    android:inputType="text" >
</EditText>
```

Em seguida, mais um `TextView` para o label do "Tipo da Viagem", e um `RadioGroup` que vai conter os `RadioButton` para fazer as opções como "Lazer" e "Negócios".

```
<TextView android:text="@string/tipo_da_viagem" />

<RadioGroup
    android:id="@+id/tipoViagem"
    android:orientation="horizontal" >

    <RadioButton
        android:id="@+id/lazer"
        android:checked="true"
        android:text="@string/lazer" />

    <RadioButton
        android:id="@+id/negocios"
        android:layout_marginLeft="30dp"
        android:text="@string/negocios" />
</RadioGroup>
```

A seguir, temos mais uma linha que conterá os labels para dois campos: "Data de Chegada" e "Data de Saída". Como não queremos que cada elemento fique em uma linha, mas sim **que os dois elementos ocupem a mesma**, vamos envolvê-los em um `TableRow` , onde cada um ocupará uma coluna:

```
<TableRow>

    <TextView
        android:layout_gravity="center"
        android:text="@string/data_da_chegada" />

    <TextView
        android:layout_gravity="center"
        android:text="@string/data_da_saida" />
</TableRow>
```

Logo abaixo do texto "Data de Chegada" e "Data de Saída", temos botões que serão utilizados para abrir caixas de diálogo específicas para seleção de datas, o `DatePickerDialog` . Também utilizaremos esse widget na tela de registro de um novo gasto. Veremos como utilizá-lo na seção *DatePicker*. Posteriormente,

revisite a ViagemActivity para implementar esses dois itens.

```
<TableRow>

    <Button
        android:id="@+id/dataChegada"
        android:onClick="selecionarData"
        android:text="@string/selecione" />

    <Button
        android:id="@+id/dataSaida"
        android:onClick="selecionarData"
        android:text="@string/selecione" />
</TableRow>
```

Por fim, montamos os campos para o valor do orçamento e quantidade de pessoas, além de um botão para confirmar o cadastro da viagem.

```
<TextView android:text="@string/orcamento" />

<EditText
    android:id="@+id/orcamento"
    android:inputType="numberDecimal" />

<TableRow>

    <TextView
        android:layout_width="wrap_content"
        android:text="@string/quantidade_de_pessoas" />

    <EditText
        android:id="@+id/quantidadePessoas"
        android:inputType="number" />
</TableRow>

<Button
    android:onClick="salvarViagem"
    android:text="@string/salvar" />
```

Com o layout pronto, crie uma nova atividade chamada ViagemActivity . Para abri-la, precisamos adicionar a implementação correta do método selecionarOpcao da DashboardActivity . O código deste método será assim:

```
public void selecionarOpcao(View view) {
    switch (view.getId()) {
        case R.id.nova_viagem:
            startActivity(new Intent(
                this, ViagemActivity.class));
            break;
    }
}
```

Quando um item for selecionado, este método será chamado, recebendo como parâmetro a `View` que foi clicada. Utilizando o `id` da `View`, saberemos qual item foi selecionado e, então, executaremos a ação desejada. Ao executar a aplicação novamente e selecionar o item "Nova Viagem", teremos como resultado uma tela semelhante à seguinte:

Figura 3.5: Tela de cadastro de uma nova viagem

3.4 DATEPICKER

O `DatePicker` é um widget projetado para a seleção de datas que pode ser utilizado tanto de maneira direta, incluindo-o diretamente no layout, como indireta, por meio de uma caixa de diálogo (`DatePickerDialog`) apresentada para o usuário.

Apesar de ser mais simples, sua utilização direta não é comum, uma vez que ele tem proporções exageradas, ocupando um espaço considerável na tela. Sua forma de uso mais comum é através do `DatePickerDialog` .

Para exemplificar o seu uso e também dar prosseguimento à implementação da nossa aplicação, vamos desenvolver o layout e a atividade referentes ao registro de um novo gasto, de acordo com o protótipo da tela de cadastro dos gastos, como na figura:

Figura 3.6: Telas de cadastro de viagem e de gasto

O arquivo XML de layout será o `gasto.xml` e a classe da atividade será a `GastoActivity`. O layout dessa tela usa elementos que já conhecemos, e é formado por um `LinearLayout` e um `TableLayout`, inseridos dentro de uma `ScrollView`, que vamos implementar aos poucos.

Primeiro podemos fazer a parte superior da tela, com um "banner" escrito "Novo Gasto" e o nome do destino, para representar seu título. Para isso, utilizaremos um `LinearLayout` aplicando uma cor diferente, com o atributo `background`, e incluiremos dois `TextViews` para as informações textuais:

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="#015488"
            android:orientation="vertical" >

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center"
                android:text="@string/novo_gasto"
                android:textAppearance=
                        "?android:attr/textAppearanceLarge"
                android:textStyle="bold" />

            <TextView
                android:id="@+id/destino"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_gravity="center"
                android:textAppearance=
                        "?android:attr/textAppearanceLarge"
                android:textStyle="bold" />
        </LinearLayout>
    </LinearLayout>
</ScrollView>
```

Em seguida, vamos fazer um `TableLayout` com duas linhas e duas colunas. Em cada coluna ficará um campo; no caso, na esquerda ficará o valor, e na direita, a data. Na linha de cima, ficará a descrição, e na inferior ficará o próprio campo. O campo para a data conterá apenas um botão, que ao ser clicado, disparará o método `selecionarData` da atividade que abre a caixa de diálogo para a seleção da data de realização do gasto.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView ...>

    <LinearLayout ...>

        <!-- Parte superior da tela -->

        <TableLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:stretchColumns="0,1">

            <TableRow>

                <TextView android:text="@string/valor" />

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_gravity="center"
                    android:text="@string/data" />
            </TableRow>

            <TableRow>

                <EditText
                    android:id="@+id/valor"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:inputType="numberDecimal" />

                <Button
                    android:id="@+id/data"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_gravity="center"
                    android:onClick="selecionarData"
                    android:text="@string/selecione" />
            </TableRow>
        </TableLayout>

        <!-- Mais campos virão aqui -->

    </LinearLayout>
</ScrollView>
```

Para implementarmos a *activity* e fazê-la abrir a caixa de diálogo para a seleção da data corretamente, no método `onCreate`

inicializamos três variáveis para representar o ano, mês e dia com base na data atual. Adicionalmente, alteramos o texto do botão para exibir a data atual e manter o usuário informado sobre a que foi selecionada.

```
public class GastoActivity extends Activity {

    private int ano, mes, dia;
    private Button dataGasto;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gasto);
        Calendar calendar = Calendar.getInstance();
        ano = calendar.get(Calendar.YEAR);
        mes = calendar.get(Calendar.MONTH);
        dia = calendar.get(Calendar.DAY_OF_MONTH);

        dataGasto = (Button) findViewById(R.id.data);
        dataGasto.setText(dia + "/" + (mes+1) + "/" + ano);
    }
}
```

O método `selecionarData` apenas invoca o método `showDialog` da própria `Activity`, passando um identificador que indica o diálogo que deve ser aberto. Essa identificação é necessária, pois podemos ter inúmeros diálogos gerenciados pela mesma *activity*. É comum encontrar códigos que fazem uso de constantes para representar os que serão abertos; no entanto, preferimos utilizar o identificador da própria `view` que deseja abrir o diálogo, uma vez que ele já é único para o layout e já está declarado.

```
public void selecionarData(View view){
    showDialog(view.getId());
}
```

Quando o método `showDialog` é invocado para criar uma caixa de diálogo pela primeira vez, o método `onCreateDialog` é chamado, passando o identificador informado, para que seja

instanciado um novo `DatePickerDialog` . Se for necessário executar alguma operação para alterar informações da caixa de diálogo sempre que ela for aberta, bastaria reescrever o método `onPrepareDialog` .

Por fim, temos que implementar um *listener* que será responsável por tratar o resultado; ou seja, a data escolhida pelo usuário. Isso se faz por meio da definição de uma classe anônima, que implementa `OnDateSetListener` , para o listener usado. Esta classe possui apenas um método que será invocado pelo próprio `DatePickerDialog` , quando uma data for selecionada, que é o `onDateSet` .

Nesse método, devemos colocar nossas regras de negócio, como a criação de uma data ou atualização de uma já existente, algum tipo de validação para verificar se a data pertence a um período válido etc. Além disso, é importante exibir para o usuário, a título de informação, qual foi a data selecionada. Em nossa implementação, recuperamos os valores de ano, mês e dia informados, e atualizamos o texto do botão para apresentar como resposta ao usuário. Agora já temos a seleção da data do gasto funcionando!

```
@Override
protected Dialog onCreateDialog(int id) {
    if(R.id.data == id){
        return new DatePickerDialog(this,
            listener, ano, mes, dia);
    }
    return null;
}

private OnDateSetListener listener = new OnDateSetListener() {
    @Override
    public void onDateSet(DatePicker view,
                int year, int monthOfYear, int dayOfMonth) {
        ano = year;
        mes = monthOfYear;
        dia = dayOfMonth;
        dataGasto.setText(dia + "/" + (mes+1) + "/" + ano);
    }
```

```
};
```

Agora que as nossas datas funcionam e já permitimos sua escolha, podemos voltar ao layout do `gasto.xml` . Ainda dentro do `LinearLayout` , adicionamos os dois campos para a descrição, e o local e um botão para cadastrar o gasto:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/descricao" />

<EditText
    android:id="@+id/descricao"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/local" />

<EditText
    android:id="@+id/local"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="text" />

<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="registrarGasto"
    android:text="@string/gastei" />
```

Figura 3.7: Seleção de datas com DatePickerDialog

Classes anônimas

Para os desenvolvedores que ainda não estão acostumados com a sintaxe do Java, é comum estranhar a definição de classes anônimas. Existem diversos artigos na internet que explicam como compreendê-las e quando usá-las, sendo que uma das explicações mais comentadas está disponível no blog da Caelum, em http://blog.caelum.com.br/classes-aninhadas-o-que-sao-e-quando-usar.

3.5 SPINNER

Na tela de registro de gastos, vamos incluir um widget para seleção de itens em uma lista suspensa. Conhecido em outras plataformas como *combo box* ou *drop-down*, no Android, ele é denominado de `spinner` .

Na nossa aplicação, gostaríamos de classificar nossos gastos

entre alimentação, hospedagem, combustível etc. Para isso, declararemos um `spinner`, e especificaremos o atributo `android:prompt` que representa o título que será apresentando, quando a lista de opções for aberta.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/categoria" />

<Spinner
    android:id="@+id/categoria"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:prompt="@string/categoria" >
</Spinner>
```

Note que, na declaração do `spinner`, não informamos quais são as opções disponíveis. Isto, porque cada item dele é uma `view` filha, proveniente de um `SpinnerAdapter`. Portanto, devemos carregar os itens previamente em um `SpinnerAdapter` e atribuí-lo ao `spinner` para que as opções possam ser exibidas.

Já estão disponíveis na plataforma Android o `ArrayAdapter` e o `SimpleAdapter`, que podem ser utilizados para criar os itens do `spinner`. Usaremos, neste momento, o `ArrayAdapter`, que permite, por exemplo, carregar diretamente uma lista de opções definidas em um arquivo de recurso. Assim, no arquivo `strings.xml`, definimos um array de `strings` com as categorias desejadas, da seguinte maneira:

```
<string-array name="categoria_gasto">
    <item>Alimentação</item>
    <item>Combustível</item>
    <item>Transporte</item>
    <item>Hospedagem</item>
    <item>Outros</item>
</string-array>
```

No método `onCreate` da `GastoActivity`, criamos um novo `ArrayAdapter` e o atribuímos ao `spinner` de categorias,

conforme o código abaixo:

```
private Spinner categoria;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.gasto);

    // busca a data atual para mostrar no botão

    ArrayAdapter<CharSequence> adapter =
        ArrayAdapter.createFromResource(
                this, R.array.categoria_gasto,
                android.R.layout.simple_spinner_item);
    categoria = (Spinner) findViewById(R.id.categoria);
    categoria.setAdapter(adapter);
}
```

Para criar o `ArrayAdapter`, utilizamos o método `createFromResource`, passando o contexto atual, o identificador do array de opções que definimos no `strings.xml` e o `id` do layout que será usado para apresentar as opções. Mais uma vez nos beneficiamos do que está disponível na plataforma e utilizamos o layout `android.R.layout.simple_spinner_item` já definido. Execute a aplicação e veja-o funcionando.

Figura 3.8: Uso do Spinner e SpinnerAdapter

3.6 LISTVIEWS

Com as telas de criação de uma nova viagem e registro de gastos criadas, precisamos de alguma forma listar as viagens e gastos realizados, para que possamos ver os detalhes e também realizar outras operações, como editar e remover. Para implementar essa funcionalidade, usaremos o widget `ListView` , que tem a capacidade de exibir itens em uma listagem. Os protótipos da nossa listagem de viagens e de gastos são os seguintes:

Figura 3.9: Listagens de viagens e gastos

Ao acessar a opção `Minhas Viagens`, será apresentada uma lista de viagens cadastradas. Ao selecionar um item da lista, o usuário poderá visualizar os gastos realizados naquela viagem em outra listagem.

Como a necessidade de criar esses tipos de listagens é bastante frequente, a plataforma Android provê algumas facilidades para a sua criação, por meio de um tipo especial de atividade, a `ListActivity`. Essa classe já possui um widget `ListView` associado, bastando que a ele seja atribuído um `ListAdapter` para prover os itens que serão exibidos na lista, seguindo a mesma ideia do `Spinner`. Com isso, podemos inclusive utilizar um `ArrayAdapter` que, além da interface `SpinnerAdapter`, também implementa `ListAdapter`.

Podemos, então, ter uma nova classe chamada `ViagemListActivity`, que, além de herdar de `ListActivity`,

também implementa OnItemClickListener , com o objetivo de tratar o evento disparado quando um item da lista é selecionado:

```
public class ViagemListActivity extends ListActivity
                                implements OnItemClickListener {
}
```

No método onCreate , temos que criar um novo ArrayAdapter , passando o layout desejado e os itens. Também recuperamos a ListView associada, através do método getListView que ganhamos por conta da herança. A essa ListView , atribuímos um listener que é a própria atividade, já que fizemos nossa classe implementar OnItemClickListener .

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setListAdapter(new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, listarViagens()));
    ListView listView = getListView();
    listView.setOnItemClickListener(this);
}

private List<String> listarViagens() {
    return Arrays.asList("São Paulo", "Bonito", "Maceió");
}
```

Repare que usamos um layout padrão do Android, android.R.layout.simple_list_item_1 , como o layout da linha da ListView . Além disso, é bem verdade que os itens da lista, ou seja, as viagens realizadas, deveriam estar armazenados em um banco de dados para serem recuperados de lá para exibição. No entanto, neste momento estamos preocupados apenas em montar as telas e as lógicas de navegação. Vamos nos preocupar com persistência no próximo capítulo, quando revisitaremos essa atividade para incluir os códigos definitivos.

Quando uma viagem da lista for selecionada, gostaríamos de navegar para a lista de gastos realizados durante aquela viagem.

Implementamos isto no método `onItemClick` , que é o método de `OnItemClickListener` invocado pela `ListView` , quando um item é escolhido. Neste momento, apresentamos a descrição do item selecionado apenas para efeito de depuração, e iniciamos a atividade referente à listagem de gastos.

```
@Override
public void onItemClick(AdapterView<?> parent, View view,
        int position, long id) {
    TextView textView = (TextView) view;
    String mensagem = "Viagem selecionada: "
        + textView.getText();

    Toast.makeText(getApplicationContext(), mensagem,
            Toast.LENGTH_SHORT).show();
    startActivity(new Intent(this, GastoListActivity.class));
}
```

Para essa listagem de gastos, precisaremos criar outra atividade, a `GastoListActivity` , que será bastante similar à que acabamos de implementar.

```
public class GastoListActivity extends ListActivity
                                implements OnItemClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setListAdapter(new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_1, listarGastos()));
        ListView listView = getListView();
        listView.setOnItemClickListener(this);
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View view,
            int position, long id) {
        TextView textView = (TextView) view;
        Toast.makeText(this,"Gasto selecionado: "
            + textView.getText(), Toast.LENGTH_SHORT).show();
    }

    private List<String> listarGastos() {
        return Arrays.asList("Sanduíche R$ 19,90",
```

```
            "Táxi Aeroporto - Hotel R$ 34,00",
            "Revista R$ 12,00");
    }
}
```

Novamente utilizamos um `ArrayAdapter` e um método para simular a recuperação dos itens que devem ser exibidos. Independentemente da viagem que foi escolhida, os gastos apresentados são sempre os mesmos. Não se preocupe com isso agora, pois vamos refatorar esses códigos e fazer o carregamento dos gastos de acordo com a viagem selecionada, **diretamente do banco de dados**. Ao executar a atividade `ViagemListActivity` e escolher um item da lista, teremos o seguinte resultado:

Figura 3.10: Listagem de viagens e gastos

Apesar de terem sido simples de implementar e serem plenamente funcionais, nossas listagens não têm um visual elegante. Porém, isso não é um problema, pois podemos customizar nossas `ListViews` e deixá-las com uma aparência mais rica. O que faremos agora é incluir uma imagem para diferenciar o tipo da viagem, assim como incluir informações do período e o total dos gastos realizados. Para a listagem de gastos, destacaremos as

categorias com cores diferentes e faremos um agrupamento por data. O protótipo a seguir serve como referência para implementação:

Figura 3.11: Protótipo de listagens personalizadas

Anteriormente, fizemos uso da `ListActivity` que já possui uma `ListView` associada, cujo layout consiste simplesmente de um `TextView` . Para listagens personalizadas, precisamos definir um layout conforme o desejado e atribuí-lo à `ListView` . Para isso, crie um novo arquivo XML de layout com o nome de `lista_viagem.xml` .

Nesse layout, vamos adicionar um `LinearLayout` com orientação horizontal, pois nossa tela é dividida em um lado esquerdo para a imagem, e o outro para os dados da viagem.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <!-- adicionaremos os outros componentes aqui -->

</LinearLayout>
```

No primeiro elemento dentro desse layout horizontal, faremos um outro `LinearLayout` , dessa vez com orientação vertical, no qual teremos a imagem de acordo com o tipo da viagem.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ...>
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:gravity="center"
        android:orientation="vertical" >

        <ImageView
            android:id="@+id/tipoViagem"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </LinearLayout>

    <!--
        ainda faltam aqui os componentes do lado direito
        com os dados das viagens
     -->

</LinearLayout>
```

Para finalizar, outro `LinearLayout` com orientação vertical com 3 `TextView` para mostrar o destino, data e gasto da viagem.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ...>

    <!-- aqui está o LinearLayout com a ImageView dentro -->

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_marginLeft="10dp"
        android:orientation="vertical" >
```

```
        <TextView
            android:id="@+id/destino"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/data"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/valor"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </LinearLayout>

</LinearLayout>
```

Quando se trabalha com listagens personalizadas, é importante definir o identificador dos widgets que vão exibir a informação, pois será necessário manipulá-los na `activity`. Repare que não definimos nenhum texto para os `TextViews` e nem uma imagem para o `ImageView`, porque seus valores serão definidos em tempo de execução, de acordo com a linha a ser exibida.

Na classe `ViagemListActivity`, substituiremos o `ArrayAdapter` que usamos no método `onCreate` e que não suporta o layout customizado que definimos, por um `SimpleAdapter`. Este *adapter* é bastante versátil, pois permite fazer uma correlação entre os dados e os widgets que devem exibi-los, permitindo o uso de layouts arbitrários tanto em `ListViews` quanto em `Spinners`.

Para criar um `SimpleAdapter`, é necessário informar um `Array` de `String` com as chaves que serão usadas para recuperar os dados, juntamente de um outro `Array` com os identificadores dos widgets (os mesmos definidos no `lista_viagem.xml`), que exibirão os dados. Os elementos dos `arrays` são correlacionados, ou seja, os dados armazenados com a chave "imagem" devem ser exibidos pelo widget com identificador `R.id.tipoViagem`. Dessa

forma, o método `onCreate` será modificado para ficar como:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String[] de = {"imagem", "destino", "data", "total"};
    int[] para = {
        R.id.tipoViagem, R.id.destino, R.id.data, R.id.valor
    };

    SimpleAdapter adapter =
        new SimpleAdapter(this, listarViagens(),
        R.layout.lista_viagem, de, para);

    setListAdapter(adapter);

    getListView().setOnItemClickListener(this);
}
```

É preciso alterar o método `listarViagens` , para retornar uma lista que será usada pelo `SimpleAdapter` , com as informações a serem exibidas. A lista é formada por um conjunto de mapas, cujas chaves devem ser os identificadores definidos no `String[] de` .

```
public class ViagemListActivity extends ListActivity
    implements OnItemClickListener {

    private List<Map<String, Object>> viagens;

    private List<Map<String, Object>> listarViagens() {
        viagens = new ArrayList<Map<String,Object>>();

        Map<String, Object> item =
            new HashMap<String, Object>();
        item.put("imagem", R.drawable.negocios);
        item.put("destino", "São Paulo");
        item.put("data","02/02/2012 a 04/02/2012");
        item.put("total","Gasto total R$ 314,98");
        viagens.add(item);

        item = new HashMap<String, Object>();
        item.put("imagem", R.drawable.lazer);
        item.put("destino", "Maceió");
        item.put("data","14/05/2012 a 22/05/2012");
        item.put("total","Gasto total R$ 25834,67");
        viagens.add(item);
```

```
        return viagens;
    }
}
```

Observe também que incluímos, para a chave `imagem`, o valor `R.drawable.negocios`, que é o identificador da imagem que deve ser utilizada pelo `ImageView`, mapeado no `int[] para` como `R.id.tipoViagem`. **Lembre-se de ter as imagens no seu diretório** `drawable`.

O `SimpleAdapter`, em tempo de execução, decide como fazer o bind dos dados informados, de acordo com o tipo de `view` fornecido. Os widgets suportados pelo `SimpleAdapter` são aqueles que implementam a interface `Checkable`, como o `TextView` e o `ImageView`. No entanto, é possível fazer a atribuição de dados para outros tipos de *views*, implementando um `ViewBinder`. Faremos algo assim para a listagem de gastos.

Por fim, faltou apenas ajustar o método `onItemClick` para recuperar a viagem, de acordo com a posição selecionada na tela, e exibir por meio do `Toast.makeText` uma informação sobre essa viagem. Com isso, o código do método ficará:

```
public class ViagemListActivity extends ListActivity
     implements OnItemClickListener {
    private List<Map<String, Object>> viagens;

    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position, long id) {
        Map<String, Object> map = viagens.get(position);
        String destino = (String) map.get("destino");
        String mensagem = "Viagem selecionada: "+ destino;
        Toast.makeText(this, mensagem,
            Toast.LENGTH_SHORT).show();
        startActivity(new Intent(this,
            GastoListActivity.class));
    }

    // métodos listarViagens e onCreate
}
```

Agora, se executarmos a aplicação, teremos a listagem personalizada de viagens.

Figura 3.12: Lista de viagens personalizada

A versão personalizada da listagem de gastos deve exibir os itens da lista em cores diferentes, de acordo com a categoria. Além disso, os gastos devem estar agrupados por data, para melhorar a organização e a usabilidade.

Continuaremos usando o `SimpleAdapter` , para alimentar a `ListView` ; e criaremos uma implementação da interface `ViewBinder` , para realizar o bind entre os dados e os widgets do layout da forma que desejamos.

O `ViewBinder` será o responsável por identificar quais gastos são da mesma data e exibir um separador na `ListView` para agrupá-los. Além disso, de acordo com a categoria, o `ViewBinder` alterará a cor de fundo do item da listagem.

Vamos criar um novo layout no arquivo `lista_gasto.xml` com a definição de um `LinearLayout` com orientação vertical, que conterá um `TextView` para mostrar a data da viagem:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/titulo"
    android:orientation="vertical" >
```

```
    <TextView
        android:id="@+id/data"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:textColor="@android:color/white"/>

</LinearLayout>
```

Vale ressaltar novamente que, como precisaremos manipular os widgets via código Java, é necessário que eles possuam um `id` , inclusive o `LinearLayout` , pois a sua cor de fundo será alterada em tempo de execução, de acordo com a categoria.

Por fim, adicionamos também outro `LinearLayout` com 2 `TextView` , para exibir a descrição e o custo da viagem.

```
<LinearLayout ...>

    <!-- TextView para a data -->

    <LinearLayout
        android:id="@+id/categoria"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >

        <TextView
            android:id="@+id/descricao"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/black" />

        <TextView
            android:id="@+id/valor"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/black" />
    </LinearLayout>

</LinearLayout>
```

Para manter nosso código organizado e facilitar alterações posteriores, vamos externalizar o esquema de cores utilizado na

listagem, assim como fazemos com as `strings` . O Android permite o uso e definição de cores em arquivos de recurso. Logo, vamos criar no diretório `res/values` um arquivo chamado `colors.xml` . As cores foram definidas da seguinte maneira:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <color name="titulo">#015488</color>
    <color name="categoria_alimentacao">#9FC5F8</color>
    <color name="categoria_hospedagem">#6FA8DC</color>
    <color name="categoria_transporte">#0099CC</color>
    <color name="categoria_outros">#ACBFD5</color>

</resources>
```

NOME DO ARQUIVO DE CORES

Não existe uma regra sobre o nome do arquivo ser `colors.xml` , você poderia usar qualquer outro. No entanto, é convencionado que um arquivo que contenha a definição das cores a serem usadas em uma aplicação Android tenha esse nome.

As cores estão no formato RGB codificadas em hexadecimal. O nome atribuído à cor se tornará o seu identificador, sendo possível acessá-la via classe `R` , como por exemplo, `R.color.categoria_alimentacao` . Vale lembrar que, assim como as `strings` e `drawables` , as cores devem ser acessadas via a API de recursos do Android.

Começamos criando os `Arrays` dentro do método `onCreate` , para utilizarmos o `SimpleAdapter` também na `GastoListActivity` :

```
public class GastoListActivity extends ListActivity
```

```
    implements OnItemClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String[] de = {
            "data", "descricao", "valor", "categoria"
        };
        int[] para = { R.id.data, R.id.descricao,
                       R.id.valor, R.id.categoria };

        SimpleAdapter adapter = new SimpleAdapter(this,
            listarGastos(), R.layout.lista_gasto, de, para);


        setListAdapter(adapter);
        getListView().setOnItemClickListener(this);
    }
}
```

Agora precisamos da implementação do método `listarGastos`, que devolve uma lista contendo mapas com as propriedades a serem usadas:

```
public class GastoListActivity extends ListActivity
    implements OnItemClickListener {

    private List<Map<String, Object>> gastos;

    // método onCreate aqui

    private List<Map<String, Object>> listarGastos() {
        gastos = new ArrayList<Map<String, Object>>();

        Map<String, Object> item =
            new HashMap<String, Object>();
        item.put("data", "04/02/2012");
        item.put("descricao", "Diária Hotel");
        item.put("valor", "R$ 260,00");
        item.put("categoria", R.color.categoria_hospedagem);
        gastos.add(item);

        // pode adicionar mais informações de viagens

        return gastos;
    }
```

```
}
```

E o método `onItemClick`, usando a nova lista de gastos para exibir a informação, quando clicada.

```
public class GastoListActivity extends ListActivity
    implements OnItemClickListener {

    // atributos, método onCreate e listarGastos

    @Override
    public void onItemClick(AdapterView<?> parent, View view,
                    int position, long id) {
        Map<String, Object> map = gastos.get(position);
        String descricao = (String) map.get("descricao");
        String mensagem = "Gasto selecionado: " + descricao;
        Toast.makeText(this, mensagem,Toast.LENGTH_SHORT).show();
    }
}
```

Agora, só nos resta fazer o `ViewBinder`, que ficará responsável por identificar os gastos da mesma data; exibir o separador na `ListView` para termos o agrupamento; e mudar a cor de fundo do item da listagem.

A criação do `ViewBinder` é feita por meio de uma nova classe que implemente essa interface, na qual se deverá implementar o método `setViewValue`. Ele retorna um `boolean` e é chamado pelo `SimpleAdapter` para cada elemento da lista informada em sua criação. Para ele, são passados três parâmetros:

- `View` — recuperada a partir de um `id` passado no `String para[]`.
- `Object` — que é o valor armazenado com a chave equivalente ao `String de[]`.
- `String` — uma representação em formato texto do dado passado (`Object data`), que será ou o resultado do método `toString()` ou uma `String` vazia, sendo garantido que seu valor jamais será nulo.

Portanto, podemos criar uma classe nova, privada, em `GastoListActivity`, que herde de `ViewBinder`:

```
public class GastoListActivity extends ListActivity
    implements OnItemClickListener {

    // atributos, método onCreate, listarGastos e onItemClick

    private class GastoViewBinder implements ViewBinder {

        @Override
        public boolean setViewValue(View view, Object data,
                String textRepresentation) {
            // vamos implementar esse método
        }
    }
}
```

Este método deve retornar `true`, caso o *bind* tenha sido realizado. Caso o retorno seja `false`, o `SimpleAdapter` tentará realizá-lo automaticamente para os tipos de *views* suportados (`Checkable`, `TextView` e `ImageView`). Se não for possível realizar o bind, uma `IllegalStateException` será lançada.

Para fazermos o agrupamento, temos que verificar qual é a `View` que está sendo processada.

Caso seja a data do gasto, então precisamos comparar também se esse gasto foi realizado na mesma data processada anteriormente ou se foi em uma nova. Sendo uma data diferente da anterior, devemos exibir o separador, que na verdade é apenas um `TextView`, cujo valor é a data do gasto. Caso sejam iguais, ou seja, os gastos foram realizados no mesmo dia, teremos que suprimir o `TextView` com `view.setVisibility(View.GONE)`, dando a impressão de agrupamento. Outra opção de visibilidade é a `View.INVISIBLE`, porém, diferentemente do `View.GONE`, a `view` não é exibida, mas continua ocupando lugar no layout.

A segunda comparação trata do `LinearLayout` que engloba

todas as informações dos gastos e se refere à categoria. Neste caso, o dado passado para o método é o próprio identificador da cor que deve ser utilizada como sua cor de fundo.

```
public class GastoListActivity extends ListActivity
    implements OnItemClickListener {

    // atributos, método onCreate, listarGastos e onItemClick
    private String dataAnterior = "";

    private class GastoViewBinder implements ViewBinder {

        @Override
        public boolean setViewValue(View view, Object data,
                String textRepresentation) {

            if(view.getId() == R.id.data){
                if(!dataAnterior.equals(data)){
                    TextView textView = (TextView) view;
                    textView.setText(textRepresentation);
                    dataAnterior = textRepresentation;
                    view.setVisibility(View.VISIBLE);
                } else {
                    view.setVisibility(View.GONE);
                }
                return true;
            }

            if(view.getId() == R.id.categoria){
                Integer id = (Integer) data;
                view.setBackgroundColor(
                    getResources().getColor(id));
                return true;
            }
            return false;
        }
    }
}
```

Por fim, agora basta indicarmos no método `onCreate` da `GastoListActivity` que o `adapter` usará o `ViewBinder` que acabamos de criar. Fazemos isso por meio do método `setViewBinder` do `adapter`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);

    String[] de = { "data", "descricao", "valor", "categoria" };
    int[] para = { R.id.data, R.id.descricao,
                   R.id.valor, R.id.categoria };

    SimpleAdapter adapter = new SimpleAdapter(this,
        listarGastos(), R.layout.lista_gasto, de, para);

    adapter.setViewBinder(new GastoViewBinder());

    setListAdapter(adapter);
    getListView().setOnItemClickListener(this);
}
```

Figura 3.13: Lista personalizada de gastos

3.7 MENUS

Os menus, da mesma maneira que os *dialogs* (como o `DatePickerDialog`), são elementos importantes de interação com

o usuário do aplicativo. Eles podem ser usados para apresentar opções globais — quando dizem respeito a uma configuração ou um estado da aplicação —, ou contextuais — quando a opção está relacionada com algum item selecionado, por exemplo.

Até a versão 3.0 do Android, era obrigatório que os aparelhos tivessem um botão *menu*, que, quando pressionado, apresentava um painel com, no máximo, seis opções visíveis. Quando existiam mais opções, era necessário selecionar uma opção "*Mais itens*" para visualizar o que restava. Esse tipo de menu é conhecido como *options menu*. Nos aparelhos mais recentes, já não existe mais o botão *menu* e muito menos botões físicos, exceto os de volume e para ligar o aparelho.

Menu de opções

Geralmente, os *options menu* são utilizados para apresentar opções que são relevantes para a *activity* atual ou para a aplicação. Criaremos, assim, alguns menus de opção para algumas de nossas atividades.

Podemos começar pela `DashboardActivity` , onde vamos criar um menu com a opção de sair da aplicação. Em seguida, para a `ViagemActivity` , criaremos um com a opção de registrar um novo gasto e apagar a viagem. Já para a `GastoActivity` , colocaremos um com a opção de remover o gasto.

Figura 3.14: Menus de opções

Os menus são definidos em arquivos XML específicos com o objetivo de externalizar a estrutura do menu do código da aplicação, além de permitir que sejam definidos diferentes menus para suportar diferentes configurações de aparelhos e versões do Android. **Os arquivos de menu devem ficar no diretório** `res/menu` . Então, vamos lá. Criaremos o menu para o dashboard no arquivo `dashbord_menu.xml` , com a seguinte definição:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/sair"
        android:icon=
            "@android:drawable/ic_menu_close_clear_cancel"
        android:title="@string/sair"/>

</menu>
```

O elemento raiz do XML é o `menu` que pode conter várias tags `item` , que representam as opções do menu. Um `item` pode ter um `id` , que será usado para saber se ele foi selecionado, além de um ícone e um título com a descrição da opção.

Os menus são efetivamente criados pela *activity* que está ativa, quando o botão *menu* é pressionado, e, nesse momento, invoca-se o método `onCreateOptionsMenu` . A própria *activity* também fornece um método para tratar, quando um item do menu for selecionado, o `onMenuItemSelected` . Vamos começar pela implementação do `onCreateOptionsMenu` .

O método `onCreateOptionsMenu` recebe como parâmetro um objeto do tipo `Menu` , no qual precisamos popular as opções, de acordo com as informações do XML que acabamos de criar, que se encontra dentro de `res/menus` . Para isso, é preciso passar os dados do XML para o objeto recebido como parâmetro, que é justamente o papel de uma classe chamada `MenuInflater` .

Recuperamos uma instância dela por meio da chamada ao método `getMenuInflater` . Em seguida, invocamos seu método `inflate` , passando como parâmetro uma referência para o menu, através da classe `R` e do objeto `menu` . Por fim, retornamos `true` para indicar que o menu deve ser exibido.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.dashboard_menu, menu);
    return true;
}
```

Como nosso menu só tem uma opção, que, no caso, é sair da aplicação, não precisamos saber qual foi o item selecionado. Então, a implementação do `onMenuItemSelected` apenas invoca o método `finish` para encerrar a atividade atual. Posteriormente neste método, também implementaremos o *logoff* do usuário.

```
    @Override
    public boolean onMenuItemSelected(int featureId, MenuItem item
) {
        finish();
        return true;
    }
```

Para a `ViagemActivity` , que é a tela utilizada para criar e editar viagens, teremos um *options menu* com as opções de registrar um novo gasto ou remover a viagem. Crie o arquivo `viagem_menu.xml` com a definição a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/novo_gasto"
        android:icon="@android:drawable/ic_menu_add"
        android:title="@string/novo_gasto"/>
    <item
        android:id="@+id/remover"
        android:icon="@android:drawable/ic_menu_delete"
        android:title="@string/remover_viagem"/>

</menu>
```

Como neste menu temos mais de uma opção, é essencial definir identificadores para seus itens, para que se possa determinar qual foi o selecionado. Para começar, faremos no `onCreateOptionsMenu` as mesmas transformações dos dados do XML para o objeto `Menu` .

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.viagem_menu, menu);
        return true;
    }
```

No método `onMenuItemSelected` , com base no `id` do item selecionado, decidimos a ação a ser executada. Quando a opção for referente ao novo gasto, iniciaremos a atividade `GastoActivity` ; e quando for a de remover, excluiremos a viagem atual. Caso não seja possível determinar qual item foi selecionado, invocamos a implementação padrão do método que retorna `false` .

```
    @Override
    public boolean onMenuItemSelected(int featureId,
            MenuItem item) {
        switch (item.getItemId()) {
```

```
        case R.id.novo_gasto:
            startActivity(new Intent(this,
                GastoActivity.class));
            return true;
        case R.id.remover:
            //remover viagem do banco de dados
            return true;
        default:
            return super.onMenuItemSelected(featureId, item);
        }
    }
```

Caso queira exercitar, crie por conta própria um menu com a opção de remover gasto para a `GastoActivity`. Essa implementação será semelhante às realizadas até o momento.

Nos *options menus*, ainda há a possibilidade de agrupar itens por meio da tag `group`, e também de criar submenus, aninhando a tag `menu`. Os grupos servem para facilitar o controle de itens que são relacionados. Através de grupos, é possível definir a visibilidade de todos os seus itens, ou se eles estão ativos ou não. O código a seguir, retirado da documentação do Android, exemplifica o uso de grupos.

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_save"
          android:icon="@drawable/menu_save"
          android:title="@string/menu_save" />
    <!-- menu group -->
    <group android:id="@+id/group_delete">
        <item android:id="@+id/menu_archive"
              android:title="@string/menu_archive" />
        <item android:id="@+id/menu_delete"
              android:title="@string/menu_delete" />
    </group>
</menu>
```

Os submenus podem ser criados simplesmente incluindo uma nova tag `menu` com seus itens associados. Isso é útil quando nossa aplicação tem um conjunto grande de opções que podem ser agrupadas em assuntos. Porém, tenha cuidado ao utilizar menus

com muitos níveis para não prejudicar a usabilidade da aplicação. Um exemplo de submenu, retirado também da documentação do Android, é o seguinte:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
          android:title="@string/file" >
        <!-- "file" submenu -->
        <menu>
            <item android:id="@+id/create_new"
                  android:title="@string/create_new" />
            <item android:id="@+id/open"
                  android:title="@string/open" />
        </menu>
    </item>
</menu>
```

Menus contextuais

Em aparelhos Android, quando queremos reenviar uma mensagem SMS para a pessoa, caso ela não tenha recebido ou tenha ocorrido uma falha no envio, não é necessário ter o trabalho de redigitá-la. O próprio aplicativo de mensagens do Android nos permite reenviar uma mensagem específica; basta pressioná-la e escolher em um menu essa opção. Esse menu que surge é específico para aquela mensagem. Chamamos isso de "Menu de Contexto"; veja um exemplo na figura:

Figura 3.15: Menu de contexto

Os menus contextuais oferecem ao usuário opções que são relevantes ao conteúdo que está sendo apresentado. Por exemplo, ao selecionar um item de uma `ListView`, podemos apresentar um menu de contexto com opções que fazem sentido para aquele item, tais como: visualizar, editar, compartilhar etc.

Para a nossa listagem de gastos, criaremos um menu contextual que permitirá ao usuário remover o gasto selecionado. A criação desse tipo de menu é similar ao que já realizamos para o menu de opções. Devemos definir um arquivo XML contendo as opções do menu, construir suas opções sobrescrevendo o método `onCreateContextMenu` e tratar a seleção do usuário no método `onContextItemSelected`.

A única diferença, além do nome dos métodos, é a necessidade de registrar a `View` que vai apresentar o menu de contexto, que fazemos por meio do método `registerForContextMenu(View`

view) . Então, no método onCreate da GastoListActivity , vamos registrar o menu de contexto.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String[] de = { "data", "descricao", "valor", "categoria" };
    int[] para = { R.id.data, R.id.descricao,
                   R.id.valor, R.id.categoria };

    SimpleAdapter adapter = new SimpleAdapter(this,
        listarGastos(), R.layout.lista_gasto, de, para);

    setListAdapter(adapter);
    getListView().setOnItemClickListener(this);

    // registramos aqui o novo menu de contexto
    registerForContextMenu(getListView());
}
```

Para a nossa listagem de gastos, como a única opção disponível no menu de contexto será a de remover, criaremos um novo XML de layout com o nome de gasto_menu.xml .

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/remover"
        android:icon="@android:drawable/ic_menu_delete"
        android:title="@string/remover"/>
</menu>
```

Fazemos o método onCreateContextMenu para ler as opções do XML e transformá-las em um objeto do tipo ContextMenu .

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
            ContextMenuInfo menuInfo) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.gasto_menu, menu);
}
```

Em seguida, precisamos reescrever o método

`onContextItemSelected` , que recebe o item do menu que foi selecionado, para realizarmos a ação adequada; no caso, fazer a remoção.

Para recuperar informações sobre o item do menu que foi selecionado, utilizamos o método `item.getMenuInfo()` , que retorna um objeto do tipo `AdapterContextMenuInfo` . Este fornece o `id` da linha que foi selecionada, a posição do item no `adapter` e a `view` que foi escolhida.

Com isso, usamos a informação da posição, para remover o item da lista de gastos. No entanto, essa operação não é refletida automaticamente na `ListView` que já renderizou as linhas na tela. É necessário fazer com que esta desenhe as linhas novamente com base no `adapter` que agora tem um item a menos. Isto é feito ao invocar-se o método `invalidateViews()` .

```
@Override
public boolean onContextItemSelected(MenuItem item) {

    if (item.getItemId() == R.id.remover) {
        AdapterContextMenuInfo info =
            (AdapterContextMenuInfo) item.getMenuInfo();
        gastos.remove(info.position);
        getListView().invalidateViews();
        dataAnterior = "";
        // remover do banco de dados
        return true;
    }
    return super.onContextItemSelected(item);
}
```

Pronto, agora temos a nossa aplicação com o menu de contextos funcionando. Em breve, integraremos com o banco de dados de verdade.

3.8 ALERTDIALOG

Além dos menus, podemos apresentar opções para o usuário

por meio de caixas de diálogo que são utilizadas geralmente para interagir com ele. Elas apresentam algum tipo de informação e solicitam que ele decida o que deve ser feito.

Podemos usar, por exemplo, uma caixa de diálogo solicitando ao usuário a confirmação da exclusão de uma informação, ou exibir uma mensagem de que determinada ação foi realizada. Os `AlertDialogs` podem conter até três botões, ou ainda uma lista de itens selecionáveis por meio de *checkboxes* ou *radio buttons*. Isso torna as formas de interação com o usuário variadas. A figura a seguir exemplifica o uso de uma caixa de diálogo para confirmar a exclusão de uma foto.

Figura 3.16: AlertDialog

Na nossa aplicação, quando o usuário selecionar uma viagem na listagem "Minhas Viagens", queremos apresentar as opções de: editar, registrar um novo gasto, visualizar os gastos já realizados e remover a viagem selecionada. Como já vimos anteriormente,

poderíamos criar um menu de contexto para resolver essa questão.

No entanto, para abrir o menu de contexto, o usuário precisa selecionar um item e mantê-lo pressionado até a exibição das opções. Isso torna a interação mais lenta e, tratando-se de funcionalidades que serão muito utilizadas, pode acabar prejudicando a usabilidade. Seria melhor se, ao selecionar um item da lista, o menu de opções fosse imediatamente apresentado. Esse comportamento é obtido ao implementarmos o menu com as opções, como uma caixa de diálogo.

O `AlertDialog` é criado por meio de um `AlertDialog.Builder`, no qual informaremos o título da caixa de diálogo e seus itens. Também é necessário fornecer um `OnClickListener` para tratar da opção escolhida pelo usuário. Nossa `ViagemListActivity` já implementa a interface `OnItemClickListener`, para capturar o clique de um item na `ListView`; agora deverá implementar também a interface `OnClickListener` e seu método `onClick`, para tratar a opção selecionada pelo usuário na caixa de diálogo.

```
public class ViagemListActivity extends ListActivity
            implements OnItemClickListener, OnClickListener {

    @Override
    public void onClick(DialogInterface dialog, int item) {
        // Vamos implementar esse método
    }
}
```

Agora vamos criar um método chamado `criaAlertDialog`, no qual definiremos um `Array` com as opções que serão exibidas na caixa de diálogo. Esse método retornará o `AlertDialog`, construído com as opções.

```
private AlertDialog criaAlertDialog() {
    final CharSequence[] items = {
            getString(R.string.editar),
            getString(R.string.novo_gasto),
```

```
            getString(R.string.gastos_realizados),
            getString(R.string.remover) };

    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle(R.string.opcoes);
    builder.setItems(items, this);

    return builder.create();
}
```

Agora, quando a `ViagemListActivity` for criada, é preciso adicionar o `AlertDialog` também. Assim, poderemos usá-lo quando for necessário. Para isso, vamos acrescentar uma chamada ao `criaAlertDialog` no `onCreate` da atividade.

```
    protected void onCreate(Bundle savedInstanceState) {
        // Realiza as outras ações

        this.alertDialog = criaAlertDialog();
    }
```

Guardamos a referência para o *dialog* criado em um atributo de instância da `ViagemListActivity`, que chamamos de `alertDialog`.

```
public class ViagemListActivity extends ListActivity
            implements OnItemClickListener, OnClickListener {

    private AlertDialog alertDialog;

    // outros atributos e métodos
}
```

No método sobrescrito `onClick`, usaremos o índice do `Array` dos itens do `AlertDialog`, para determinar qual opção foi selecionada, e executaremos a ação apropriada. Para que o `AlertDialog` funcione de forma similar a um menu de contexto e também tenha a opção para realizar a exclusão da viagem, precisaremos manter uma referência para o item da `ListView` que foi selecionado. Isso é feito no método sobrescrito `onItemClick`, que armazena a posição do item selecionado e, só então, abre a caixa de diálogo, invocando `alertDialog.show()`. Quando a opção

"remover" da caixa de diálogo é selecionada, utilizamos a posição previamente armazenada para remover a viagem da `ListView`.

```
@Override
public void onClick(DialogInterface dialog, int item) {
    switch (item) {
    case 0:
        startActivity(new Intent(this,
            ViagemActivity.class));
        break;
    case 1:
        startActivity(new Intent(this,
            GastoActivity.class));
        break;
    case 2:
        startActivity(new Intent(this,
            GastoListActivity.class));
        break;
    case 3:
        viagens.remove(this.viagemSelecionada);
        getListView().invalidateViews();
        break;
    }
}

@Override
public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {
    this.viagemSelecionada = position;
    alertDialog.show();
}
```

Note que precisamos ter um novo atributo na `ViagemListActivity` para guardar a viagem selecionada.

```
private int viagemSelecionada;
```

Pronto. Agora temos uma versão de menu de contexto muito mais responsiva. Execute a aplicação e confira!

Figura 3.17: Menu utilizando AlertDialog

AlertDialogs com confirmações

Geralmente, operações críticas da aplicação requerem a confirmação do usuário. É o caso, por exemplo, da exclusão de uma viagem ou de um gasto realizado. As caixas de diálogo de confirmação, com botões "sim/não", são implementadas utilizando `AlertDialogs` . Vamos alterar nosso código para solicitar a confirmação do usuário, para remover uma viagem. A viagem será removida só após essa confirmação.

Será necessário criar um novo diálogo que inclua os botões para a confirmação ou rejeição, e passar um *listener* para tratar qual botão foi escolhido. Como nossa atividade já implementa `onClickListener` , usaremos o método já existente como *listener* do diálogo de confirmação.

Vamos fazer o método `criaDialogConfirmacao` que criará o novo *dialog*, com as opções de confirmação e negação.

```
private AlertDialog criaDialogConfirmacao() {
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setMessage(R.string.confirmacao_exclusao_viagem);
    builder.setPositiveButton(getString(R.string.sim), this);
    builder.setNegativeButton(getString(R.string.nao), this);
```

```
    return builder.create();
}
```

Repare na invocação dos métodos `setPositiveButton` e `setNegativeButton` , que fazem os botões "*Sim*" e "*Não*", respectivamente, e que recebem a `string` que deve ser exibida para cada um.

O próximo passo é invocar o `criaDialogConfirmacao` no `onCreate` da *activity*, e guardar uma referência para o novo `AlertDialog` em um atributo de instância:

```
// Novo atributo de instância
private AlertDialog dialogConfirmacao;

protected void onCreate(Bundle savedInstanceState) {
    // Realiza as outras ações

    this.alertDialog = criaAlertDialog();
    this.dialogConfirmacao = criaDialogConfirmacao();
}
```

No método `onClick` , quando a opção "remover" for selecionada, o *dialog* de confirmação será exibido. Adicionamos as verificações para determinar se o botão pressionado foi referente ao "*Sim*" ou ao "*Não*". Caso a escolha tenha sido negativa, a remoção não é confirmada e nada deve ser feito, exceto fechar a caixa de diálogo. Isso que pode ser realizado através do método `dismiss()` do próprio `AlertDialog` .

```
    @Override
    public void onClick(DialogInterface dialog, int item) {
        switch (item) {
        case 0:
            startActivity(
                new Intent(this, ViagemActivity.class));
            break;
        case 1:
            startActivity(
                new Intent(this, GastoActivity.class));
            break;
        case 2:
```

```
            startActivity(
                new Intent(this, GastoListActivity.class));
            break;
        case 3:
            dialogConfirmacao.show();
            break;
        case DialogInterface.BUTTON_POSITIVE:
            viagens.remove(viagemSelecionada);
            getListView().invalidateViews();
            break;
        case DialogInterface.BUTTON_NEGATIVE:
            dialogConfirmacao.dismiss();
            break;
    }
}
```

O `AlertDialog` criado ficará assim:

Figura 3.18: AlertDialog com confirmação

3.9 PROGRESSDIALOG E PROGRESSBAR

Sempre que uma operação demorada for executada — como fazer download de informações da Internet —, é importante manter o usuário informado sobre o que está acontecendo e que a aplicação continua funcionando. Nessas situações, podemos utilizar um `ProgressDialog` , uma extensão do `AlertDialog` , e apresentar ao usuário uma animação representando o progresso da operação. Também é possível informar um título e uma mensagem para exibição, além de botões para controlar a operação, se necessário.

O `ProgressDialog` pode ter uma duração indeterminada, quando não há previsão de término, ou determinada; quando a duração tem um valor conhecido. No primeiro caso, é apresentado ao usuário uma animação em formato de círculo.

Quando o `ProgressDialog` tem duração determinada, uma barra de progresso é apresentada e é possível acompanhar o andamento da tarefa por meio do valor ou da porcentagem. Confira na figura a seguir alguns exemplos de uso:

Figura 3.19: Exemplos de ProgressDialog

O código a seguir exemplifica como exibir um `ProgressDialog` com duração indeterminada e que pode ser

cancelado pelo usuário, quando o botão "*Voltar*" do aparelho for pressionado:

```
boolean podeCancelar = true;
boolean indeterminado = true;
String titulo = "Operação demorada";
String mensagem = "Aguarde...";
ProgessDialog dialog = ProgressDialog.show(this, titulo,
    mensagem, indeterminado, podeCancelar);
```

Quando precisamos indicar o progresso da operação para o usuário, a forma de criar o `ProgressDialog` é um pouco diferente.

Utilizamos o construtor `ProgressDialog(Context)` e definimos o título e a mensagem por meio de *setters*. É necessário definir um estilo para a barra de progresso, por exemplo `ProgressDialog.STYLE_HORIZONTAL` . Também podemos definir o valor máximo dessa barra que, ao ser alcançado, provoca o fechamento da caixa de diálogo. A atualização do progresso pode ser realizada através dos métodos `setProgress(int)` ou `incrementProgressBy(int)` .

Veja um código de exemplo que também inclui botões:

```
ProgressDialog dialog = new ProgressDialog(this);
dialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
dialog.setTitle(titulo);
dialog.setMessage(mensagem);
dialog.setCancelable(true);
dialog.setMax(800);
dialog.setButton(Dialog.BUTTON_NEGATIVE,
                        getString(R.string.abortar), this);
dialog.setButton(Dialog.BUTTON_NEUTRAL,
                        getString(R.string.pausar), this);
dialog.setButton(Dialog.BUTTON_POSITIVE,
                        getString(R.string.continuar), this);

dialog.show();
```

Em algumas situações, há a possibilidade de não termos uma operação em andamento, porém queremos indicar para o usuário o quão próximo ele está de alcançar determinado valor limite. Na

nossa aplicação, podemos, por exemplo, usar uma `ProgressBar` para exibir o total de gastos realizados em relação ao orçamento estipulado para a viagem.

Incluiremos essa funcionalidade na listagem de viagens, e consideraremos que existe um valor limite configurado para os gastos que pode ser maior ou menor do que o valor do orçamento. Posteriormente, criaremos essa configuração e notificaremos o usuário quando esse valor limite for alcançado.

A `ProgressBar` possui dois valores de progresso, um principal e outro secundário. Utilizaremos o principal para exibir os valor total dos gastos realizados, e o secundário para marcar o valor limite, se este for menor do que o orçamento. Inclua no layout `lista_viagem.xml` a definição da barra de progresso abaixo do `TextView` que exibe o valor dos gastos realizados:

```
<ProgressBar android:id="@+id/barraProgresso"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            style="?android:attr/progressBarStyleHorizontal"/>
```

Depois, precisaremos alterar a `ViagemListActivity` para implementar `ViewBinder`, pois agora temos uma `ProgressBar` na `ListView`. Para preencher os valores da barra de progresso, passaremos um `array` com os valores definidos para o orçamento, o limite e o valor dos gastos, para o mapa usado pelo `SimpleAdapter`.

O primeiro passo, no método `onCreate`, é adicionar um novo item nos `Arrays` `de` e `para`, que vão referenciar a barra de progresso que acabamos de definir no layout.

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String[] de = { "imagem", "destino", "data",
                        "total", "barraProgresso" };
```

```
    int[] para = { R.id.tipoViagem, R.id.destino,
            R.id.data, R.id.valor, R.id.barraProgresso };

    // restante da implementação
}
```

Em seguida, no método `listarViagens`, que devolve as informações das viagens realizadas, vamos devolver as informações necessárias para a barra, adicionando ao `Map` uma nova informação:

```
private List<Map<String, Object>> listarViagens() {
    viagens = new ArrayList<Map<String, Object>>();

    Map<String, Object> item = new HashMap<String, Object>();
    item.put("imagem", R.drawable.negocios);
    item.put("destino", "São Paulo");
    item.put("data", "02/02/2012 a 04/02/2012");
    item.put("total", "Gasto total R$ 314,98");
    item.put("barraProgresso",
        new Double[]{ 500.0, 450.0, 314.98});
    viagens.add(item);

    // adiciona mais informações se preferir

    return viagens;
}
```

Por fim, temos que sobrescrever o método `setViewValue`, no qual são atribuídos os valores de progresso principal e secundário, além do valor máximo da barra de progresso, representado aqui pelo valor definido como o orçamento disponível para a realização da viagem.

```
@Override
public boolean setViewValue(View view, Object data,
            String textRepresentation) {
    if (view.getId() == R.id.barraProgresso) {
        Double valores[] = (Double[]) data;
        ProgressBar progressBar = (ProgressBar) view;
        progressBar.setMax(valores[0].intValue());
        progressBar.setSecondaryProgress(
            valores[1].intValue());
        progressBar.setProgress(
```

```
                valores[2].intValue());
            return true;
        }
        return false;
    }
```

A nova listagem de viagens exibindo a `ProgressBar` ficará assim:

Figura 3.20: Listagem de viagens com ProgressBar

3.10 PREFERÊNCIAS

Muitas vezes, é interessante que o usuário configure o comportamento do aplicativo, de acordo com a sua preferência. Isso torna-o muito mais atrativo e personalizado. Mas é claro que o esforço para desenvolver um aplicativo altamente customizável é enorme. É necessário parcimônia e oferecer ao usuário algumas

opções-chave. O Android oferece suporte para a gravação e armazenamento dessas preferências, bem como uma `PreferenceActivity` , para permitir a sua edição.

As preferências são armazenadas em pares com um elemento representando a chave, que servirá para a sua obtenção posterior, e outro representando o valor da preferência. Podemos utilizar esse tipo de armazenamento não só para as preferências do usuário, mas também para qualquer outro tipo de dado básico que a aplicação possa necessitar, como por exemplo, o endereço de um serviço remoto ou informações carregadas da Internet.

Existe um arquivo de preferências padrão que pode ser usado pela aplicação e, quando necessário, é possível criar vários arquivos de preferências para armazenar informações distintas. Eles também podem ser criados por `Activity` .

Na nossa aplicação, disponibilizaremos duas opções de preferência para o usuário: uma para que ele informe um valor percentual do orçamento das viagens, que, quando ultrapassado pelo total de gastos realizados, provocará uma notificação; e outra que configure o aplicativo em um "Modo Viagem". A proposta é que, quando este modo estiver selecionado e o usuário for registrar um novo gasto, o aplicativo selecione automaticamente, com base na data atual, a viagem da qual ele se refere, em vez de apresentar uma listagem para que o usuário selecione a viagem correta.

Como o Android já facilita a implementação de telas de preferências, disponibilizando elementos que já tratam da apresentação e gravação dos itens, criaremos um arquivo XML diferente dos arquivos de layout criados até o momento. Por questões de organização, manteremos o arquivo XML das preferências no diretório `res/xml` . Crie nessa pasta o arquivo `preferencias.xml` , com a seguinte definição:

```xml
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">

    <PreferenceCategory android:title="@string/preferencias">
        <CheckBoxPreference
            android:key="modo_viagem"
            android:summary="@string/modo_viagem_sumario"
            android:title="@string/modo_viagem" />

        <EditTextPreference
            android:dialogTitle="@string/informe_valor_limite"
            android:key="valor_limite"
            android:defaultValue="80"
            android:summary="@string/valor_limite_sumario"
            android:title="@string/valor_limite"/>
    </PreferenceCategory>

</PreferenceScreen>
```

Repare que não há nenhum layout ou widget declarado, como o elemento raiz o `<PreferenceScreen>` . Os elementos que compõem a tela de preferências são específicos. As preferências podem ser agrupadas em categorias, por meio do elemento `<PreferenceCategory>` , para que sejam apresentadas visualmente no mesmo grupo. Em seguida, definimos uma preferência que faz uso de um *checkbox*, através do `CheckBoxPreference` , para configurar o aplicativo em "Modo Viagem". Já para a definição do valor limite de gastos, utilizamos uma preferência associada a um `EditText` , cujo valor padrão é 80.

O valor informado no atributo `key` será o identificador da chave, na qual será armazenada a preferência e utilizada posteriormente para a recuperação do valor gravado. Para os atributos `title` , `summary` e `dialogTitle` , definimos mensagens que serão apresentadas para o usuário com a descrição da preferência que está sendo configurada. Agora, precisamos criar uma *activity* que estende a classe `PreferenceActivity` , e carregar o XML que define as opções. Crie uma classe chamada `ConfiguracoesActivity` com o seguinte código:

```java
public class ConfiguracoesActivity extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferencias);
    }
}
```

No método `onCreate`, invocamos o `addPreferencesFromResource` disponível na `PreferenceActivity` para carregar o arquivo XML de preferências e construir a tela. Quando a opção "Configurações" da `DashboardActivity` for escolhida, a tela de preferências criada deverá ser exibida.

Faça as alterações necessárias no método `selecionarOpcao` para abrir a tela de preferências (`ConfiguracoesActivity`) e confira como ela ficou.

```java
public void selecionarOpcao(View view) {
    switch (view.getId()) {
    //códigos existentes
    case R.id.configuracoes:
        startActivity(
            new Intent(this, ConfiguracoesActivity.class));
        break;
    }
}
```

Figura 3.21: Telas de preferências do aplicativo

Neste momento, não implementaremos as funcionalidades que vão utilizar essas preferências que acabamos de armazenar, pois dependemos de itens que ainda não estão prontos. No entanto, para exercitar seu uso, vamos implementar outra funcionalidade, que é incluir na nossa aplicação uma opção para o usuário manter-se logado no aplicativo. Isso utilizará a classe `SharedPreferences` , para acessar e gravar dados em um arquivo de preferência de uma `Activity` .

Na tela de login, incluiremos um *checkbox* para indicar a opção do usuário e gravaremos essa informação no arquivo de preferências da atividade. Quando o aplicativo for iniciado, a preferência será consultada para determinar se a tela de login deve ser apresentada ou se a dashboard deve ser exibida. Altere o `login.xml` para incluir uma *checkbox* da seguinte forma e verifique como ficou o código da `BoaViagemActivity` :

```
<CheckBox
        android:id="@+id/manterConectado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/manter_conectado" />
```

```
public class BoaViagemActivity extends Activity {
    private static final String MANTER_CONECTADO =
        "manter_conectado";
    private EditText usuario;
    private EditText senha;
    private CheckBox manterConectado;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);

        usuario = (EditText) findViewById(R.id.usuario);
        senha = (EditText) findViewById(R.id.senha);
        manterConectado =
            (CheckBox) findViewById(R.id.manterConectado);

        SharedPreferences preferencias =
            getPreferences(MODE_PRIVATE);
        boolean conectado =
            preferencias.getBoolean(MANTER_CONECTADO, false);

        if(conectado){
            startActivity(
                new Intent(this, DashboardActivity.class));
        }
    }
}
```

Precisamos obter uma instância de `SharedPreferences` no modo privado, o que permite alterações no arquivo de preferência apenas pela aplicação que a criou. Em seguida, tentaremos recuperar algum valor gravado para a chave `manter_conectado`. Caso nenhum valor seja encontrado, o valor `false` deve ser retornado. Caso o recuperado seja `true`, em vez de apresentar a tela de login, a `DashboardActivity` é iniciada. No método `entrarOnClick`, tratamos da escolha do usuário em se manter ou não conectado.

```java
    public void entrarOnClick(View v) {
        String usuarioInformado = usuario.getText().toString();
        String senhaInformada = senha.getText().toString();

        if("leitor".equals(usuarioInformado) &&
            "123".equals(senhaInformada)) {

            SharedPreferences preferencias =
                getPreferences(MODE_PRIVATE);

            Editor editor = preferencias.edit();
            editor.putBoolean(MANTER_CONECTADO,
                manterConectado.isChecked());
            editor.commit();

            startActivity(
                new Intent(this,DashboardActivity.class));
        }
        else{
            String mensagemErro =
                getString(R.string.erro_autenticacao);
            Toast toast = Toast.makeText(this, mensagemErro,
                Toast.LENGTH_SHORT);
            toast.show();
        }
    }
}
```

Após a autenticação bem sucedida, podemos recuperar um `Editor` para fazer as alterações desejadas no arquivo de preferências, e o usamos para incluir na chave `manter_conectado` o valor booleano obtido da *checkbox*. Para efetivar as alterações, é necessário invocar o método `editor.commit()` . Em seguida, iniciamos a dashboard. Execute a aplicação e experimente essa nova funcionalidade!

Para definir outros arquivos de preferências ou ter um arquivo de preferência que será acessado por mais atividades a partir de um determinado nome, basta recuperar uma `SharedPreferences` desta forma:

```java
String NOME_PREFERENCIAS = "PREFERENCIAS_BOAVIAGEM";
SharedPreferences preferencias =
        getSharedPreferences(NOME_PREFERENCIAS, 0);
```

Quando utilizamos as facilidades da `PreferenceActivity` , o arquivo de preferências criado por ela deve ser acessado por meio do `PreferenceManager` , informando o contexto (a atividade que irá utilizar o arquivo) da seguinte forma:

```
SharedPreferences preferencias = PreferenceManager
        .getDefaultSharedPreferences(contexto);
```

3.11 CONCLUSÃO

Chegamos ao final de mais um capítulo! Aqui aprendemos na prática como utilizar os principais layouts disponíveis na plataforma Android, bem como fazer uso dos widgets fundamentais para a entrada de dados. Também criamos `ListViews` personalizadas para exibir listagens e acrescentamos funcionalidades, como menus de opção e de contexto. Empregamos `AlertDialogs` para a aplicação se comunicar com o usuário, além de salvar as suas preferências.

A primeira versão da nossa aplicação BoaViagem está quase pronta! Vamos prosseguir para o próximo capítulo, para descobrir como persistir e recuperar os dados usando o SQLite.

CAPÍTULO 4

PERSISTÊNCIA DE DADOS NO ANDROID COM SQLITE

Para o nosso aplicativo BoaViagem, as interações com o usuário já estão praticamente prontas; no entanto, ainda não estamos persistindo os dados das viagens e dos gastos realizados. O objetivo deste capítulo é armazenar e recuperar dados da nossa aplicação, utilizando o SQLite, disponível na plataforma Android, que, ao contrário da maioria dos bancos de dados SQL, não necessita de um processo servidor.

O SQLite armazena as tabelas, as views, os índices e as *triggers* em apenas um arquivo em disco, no qual são realizadas as operações de leitura e escrita. No Android, o banco de dados é acessível por qualquer classe da aplicação que o criou, mas não pode ser acessado por outra. Quando a aplicação que contém o banco de dados é desinstalada, os dados armazenados também são removidos.

4.1 O PROCESSO DE CRIAÇÃO DO BANCO DE DADOS

Para utilizar o SQLite em nossa aplicação, precisamos usar uma API que já possua todo o trabalho de se comunicar com o banco de dados encapsulado dentro dela. É justamente esse o papel da classe `SQLiteOpenHelper` , que devemos herdar.

Essa classe facilita a criação, o versionamento e o acesso ao

banco de dados. Então, vamos começar! Crie uma nova classe com o nome de `DatabaseHelper` , herdando de `SQLiteOpenHelper` .

```
public class DatabaseHelper extends SQLiteOpenHelper{
...
}
```

Ao herdarmos dessa classe, devemos implementar os métodos `onCreate` e `onUpgrade` para criar as tabelas e incluir os dados iniciais, caso necessário. Além disso, é preciso tratar das regras de atualização de dados e de estrutura do banco, quando necessário.

Também precisaremos chamar o construtor do `SQLiteOpenHelper` , informando o contexto, o nome do banco de dados, sua versão atual e, se necessário, um `CursorFactory` :

```
public class DatabaseHelper extends SQLiteOpenHelper{

    private static final String BANCO_DADOS = "BoaViagem";
    private static int VERSAO = 1;

    public DatabaseHelper(Context context) {
        super(context, BANCO_DADOS, null, VERSAO);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {}

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
            int newVersion) {}
}
```

Teremos duas tabelas no banco de dados: uma para armazenar os dados das viagens e outra para os gastos realizados. Sua criação deve ser feita no método `onCreate` , que é invocado quando tentamos acessar o banco de dados pela primeira vez e ele ainda não está criado.

Para criar as tabelas, executamos uma instrução SQL, usando o método `execSQL` da classe `SQLiteDatabase` que não possui

retorno. Dessa forma, podemos utilizá-lo somente para instruções, cujo resultado não precisa ser avaliado. Já o método `onUpgrade` deve implementar as regras para atualização da estrutura do banco e também dos dados, quando uma nova versão for disponibilizada.

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE viagem (_id INTEGER PRIMARY KEY," +
        " destino TEXT, tipo_viagem INTEGER, " +
        " data_chegada DATE, data_saida DATE, " +
        " orcamento DOUBLE, quantidade_pessoas INTEGER);");

    db.execSQL("CREATE TABLE gasto (_id INTEGER PRIMARY KEY," +
        " categoria TEXT, data DATE, valor DOUBLE," +
        " descricao TEXT, local TEXT, viagem_id INTEGER," +
        " FOREIGN KEY(viagem_id) REFERENCES viagem(_id));");
}
```

Quando uma nova versão do aplicativo e do banco de dados for lançada, o Android verificará qual versão do banco de dados o usuário possui; se esta for menor do que a atual, o método `onUpgrade` será invocado. Como, por enquanto, ainda estamos na primeira versão, não implementaremos nada no método `onUpgrade` .

Caso seja necessário, por exemplo, em uma nova versão do aplicativo, acrescentar uma nova coluna na tabela `gasto` para armazenar o nome da pessoa que o realizou, poderíamos implementar o método `onUpgrade` dessa forma:

```
@Override
public void onUpgrade(SQLiteDatabase db,
                      int oldVersion, int newVersion) {

    db.execSQL("ALTER TABLE gasto ADD COLUMN pessoa TEXT");

}
```

Tanto a criação do banco quanto a sua atualização só acontecem de fato, quando obtemos uma instância de `SQLiteDatabase` , e não quando instanciamos o `DatabaseHelper` . Veremos esses detalhes

na sequência.

Convenção para a coluna ID

Um detalhe importante é que as colunas que são chaves primárias possuem o nome de `_id` . Esta é uma convenção usada no Android, para que os resultados de consultas realizadas nessas tabelas possam ser utilizadas em `CursorAdapters` , que dependem de uma coluna com este nome.

4.2 GRAVAÇÃO DAS VIAGENS NO BANCO DE DADOS

Com o banco de dados já preparado, vamos começar a armazenar os dados das viagens. Neste momento, focaremos em como realizar as operações no banco de dados. No método `onCreate` da `ViagemActivity` , instanciaremos o `DatabaseHelper` e também criaremos referências para todas as `views` que contém os dados informados pelo usuário, pois teremos que os gravar. O código ficará assim:

```
// novos atributos
private DatabaseHelper helper;
private EditText destino, quantidadePessoas, orcamento;
private RadioGroup radioGroup;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.viagem);

    Calendar calendar = Calendar.getInstance();
    ano = calendar.get(Calendar.YEAR);
    mes = calendar.get(Calendar.MONTH);
    dia = calendar.get(Calendar.DAY_OF_MONTH);
```

```
    dataChegadaButton = (Button) findViewById(R.id.dataChegada);
    dataSaidaButton = (Button) findViewById(R.id.dataSaida);

    // recuperando novas views
    destino = (EditText) findViewById(R.id.destino);
    quantidadePessoas =
        (EditText) findViewById(R.id.quantidadePessoas);
    orcamento = (EditText) findViewById(R.id.orcamento);
    radioGroup = (RadioGroup) findViewById(R.id.tipoViagem);

    // prepara acesso ao banco de dados
    helper = new DatabaseHelper(this);
}
```

Agora, precisaremos implementar, de fato, a inserção dos dados informados pelo usuário. Isto será feito no método `salvarViagem`, que é disparado quando o usuário pressiona o botão "Salvar". Para realizar operações de escrita no banco de dados, devemos recuperar um `SQLiteDatabase` por meio do método `getWritableDatabase()`, definido na classe `SQLiteOpenHelper` e disponível por meio de herança na nossa classe `DatabaseHelper`.

```
public void salvarViagem(View view){
    SQLiteDatabase db = helper.getWritableDatabase();
}
```

Para inserir os dados, podemos montar manualmente uma instrução SQL de `insert`, ou utilizar o `ContentValues`, informando um conjunto de dados no formato chave-valor, no qual a chave é a coluna do banco de dados e o valor é o dado a ser armazenado.

Por simplificação, não realizaremos nenhuma validação dos dados, simplesmente recuperaremos os informados pelo usuário por meio das `views`, e colocaremos em um `ContentValues`:

```
public void salvarViagem(View view) {
    SQLiteDatabase db = helper.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put("destino", destino.getText().toString());
```

```
    values.put("data_chegada", dataChegada.getTime());
    values.put("data_saida", dataSaida.getTime());
    values.put("orcamento", orcamento.getText().toString());
    values.put("quantidade_pessoas",
                    quantidadePessoas.getText().toString());

    int tipo = radioGroup.getCheckedRadioButtonId();

    if(tipo == R.id.lazer) {
        values.put("tipo_viagem", Constantes.VIAGEM_LAZER);
    } else {
        values.put("tipo_viagem", Constantes.VIAGEM_NEGOCIOS);
    }
}
```

Agora que temos o `ContentValues` preparado, podemos invocar o método `insert` do `SQLiteDatabase` , que receberá 3 parâmetros. O primeiro informará a tabela, e o terceiro receberá o `ContentValues` . Já o segundo parâmetro, em que informaremos `null` , representará o nome das chaves do `ContentValues` que devem ter seu valor inserido como `null` . No nosso caso, não desejamos que nenhuma coluna tenha seu valor anulado, por isso não passamos nada.

Por fim, caso o registro seja inserido com sucesso, o método `insert` retorna o identificador do novo registro e, em caso de falha, retorna `-1` . Mostramos uma mensagem na tela de acordo com esse resultado.

```
public void salvarViagem(View view) {
    // prepara o ContentValues

    long resultado = db.insert("viagem", null, values);

    if(resultado != -1 ){
        Toast.makeText(this, getString(R.string.registro_salvo),
                Toast.LENGTH_SHORT).show();
    }else{
        Toast.makeText(this, getString(R.string.erro_salvar),
                Toast.LENGTH_SHORT).show();
    }
}
```

A coluna `tipo_viagem` é do tipo `Integer` e, para evitar o uso direto de números no código, podemos criar constantes para representar os seus possíveis valores. Como isto é recorrente, criaremos uma classe chamada `Constantes` que vai centralizar todas as constantes usadas na aplicação. Inicialmente, essa classe terá apenas os tipos de viagem, mas, no futuro, adicionaremos novas informações. O código dela é o seguinte:

```
public class Constantes {
    public static final int VIAGEM_LAZER = 1;
    public static final int VIAGEM_NEGOCIOS = 2;
}
```

O leitor mais atento deve ter percebido que, ao incluir os valores no `ContentValues` para serem inseridos no banco de dados, o tipo do dado não é o mesmo daquele determinado na criação da tabela correspondente. Por exemplo, a coluna `data_chegada` é do tipo `Date`, porém atribuímos a ela um valor do tipo `long`.

Não há problema! O SQLite trabalha com tipos dinâmicos e sabe converter os dados adequadamente para o formato desejado. Vale ressaltar que, para os tipos `Date`, é interessante armazená-los informando o seu valor como `long`, para que possamos posteriormente construir mais facilmente um objeto do tipo `java.util.Date`.

Um detalhe importante é a necessidade de fechar o banco de dados, quando seu uso não for mais necessário. Podemos fazer isso chamando o método `helper.close()`. Nem sempre é fácil determinar o melhor momento para fechá-lo, por isso é comum o fazer quando a *activity* for finalizada e destruída, sobrescrevendo o método `onDestroy`. **Lembre-se de fazer isso sempre que utilizar um banco de dados**.

```
@Override
protected void onDestroy() {
    helper.close();
    super.onDestroy();
```

```
}
```

Com essas alterações, finalizamos a inserção das informações de viagens! Seus dados agora já são armazenados no banco de dados. Fácil, não é mesmo? Agora, precisamos saber quais deles temos gravados.

4.3 LISTANDO AS VIAGENS DIRETO DO SQLITE

Uma vez que as informações das viagens já estão armazenadas no banco de dados, precisaremos recuperá-las para a exibição na lista de viagens, que no momento conta apenas com uma implementação temporária e dados estáticos. Na classe `ViagemListActivity` , vamos reaproveitar os códigos que tratam da `ListView` e alterar a implementação do método `listarViagens()` , para buscar as informações das viagens no banco de dados.

Para iniciar essa implementação, devemos instanciar um `DatabaseHelper` e recuperar o valor limite de gastos das preferências do usuário. Podemos aproveitar e criar um `SimpleDateFormat` para formatar as datas recuperadas do banco de dados, tudo isso no método `onCreate` .

```
private DatabaseHelper helper;
private SimpleDateFormat dateFormat;
private Double valorLimite;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    helper = new DatabaseHelper(this);
    dateFormat = new SimpleDateFormat("dd/MM/yyyy");

    SharedPreferences preferencias =
            PreferenceManager.getDefaultSharedPreferences(this);
```

```
    String valor = preferencias.getString("valor_limite", "-1");
    valorLimite = Double.valueOf(valor);

    //códigos existentes
}
```

No método listarViagens() , faremos uma consulta no banco de dados para obter todas as viagens cadastradas. Ao contrário do que foi feito para salvar um registro, que é uma operação de escrita, agora utilizaremos uma instância de leitura do SQLiteDatabase , por meio do método getReadableDatabase . A partir dela, faremos uma consulta e obteremos um Cursor para navegar pelos resultados.

```
private List<Map<String, Object>> listarViagens() {

    SQLiteDatabase db = helper.getReadableDatabase();
    Cursor cursor =
            db.rawQuery("SELECT _id, tipo_viagem, destino, " +
        "data_chegada, data_saida, orcamento FROM viagem",null);
    //códigos existentes
}
```

Nessa implementação, usamos o método rawQuery que executa um SQL diretamente. O Cursor retornado está sempre posicionado antes do primeiro resultado. Para iniciarmos a iteração sobre os dados, precisamos apontá-lo para o primeiro registro e também saber a quantidade de linhas retornadas. Além disso, utilizaremos um método para avançar para o próximo registro e para fechar o cursor , quando finalizarmos a iteração. Os métodos disponíveis para isso são os seguintes:

- cursor.moveToFirst() — move o cursor para o primeiro registro;
- cursor.getCount() — retorna a quantidade de linhas;
- cursor.moveToNext() — avança para o próximo registro;
- cursor.close() — fecha o cursor.

Para obter os dados do `Cursor` , devemos invocar um *getter* com o tipo do dado, informando o índice da coluna desejada. Por exemplo, para recuperar o valor da coluna `_id` , que é do tipo `INTEGER` , chamamos o método `cursor.getInt(0)` , onde `0` é o índice da coluna. Os outros getters disponíveis são:

- `cursor.getInt(columnIndex)`
- `cursor.getFloat(columnIndex)`
- `cursor.getLong(columnIndex)`
- `cursor.getShort(columnIndex)`
- `cursor.getString(columnIndex)`
- `cursor.getBlob(columnIndex)`

Quando não sabemos ao certo o índice da coluna, mas sabemos o seu nome, podemos utilizar o método `getColumnIndex(columnName)` para recuperar sua posição.

Agora que realizamos a consulta e recuperamos o `Cursor` , precisamos posicioná-lo no primeiro registro, por meio do método `moveToFirst` .

```
SQLiteDatabase db = helper.getReadableDatabase();
Cursor cursor =
        db.rawQuery("SELECT _id, tipo_viagem, destino, " +
            "data_chegada, data_saida, orcamento FROM viagem",
            null);

cursor.moveToFirst();
```

O próximo passo é guardar em um mapa os dados de cada registro. Nesse mapa, a chave conterá o nome da informação e o valor será a informação que foi recuperada do banco de dados. Teremos que fazer isso para `id` , `tipo de viagem` , `destino` , `data de chegada` , `data de saida` e `orcamento` , fazendo conversões onde necessário, como por exemplo, de `long` para `Date` . Todos os registros serão guardados em um `ArrayList` , contendo as viagens:

```java
private List<Map<String, Object>> listarViagens() {
    SQLiteDatabase db = helper.getReadableDatabase();
    Cursor cursor =
        db.rawQuery("SELECT _id, tipo_viagem, destino, " +
            "data_chegada, data_saida, orcamento FROM viagem",
            null);

    cursor.moveToFirst();


    viagens = new ArrayList<Map<String, Object>>();

    for (int i = 0; i < cursor.getCount(); i++) {

        Map<String, Object> item =
            new HashMap<String, Object>();

        String id = cursor.getString(0);
        int tipoViagem = cursor.getInt(1);
        String destino = cursor.getString(2);
        long dataChegada = cursor.getLong(3);
        long dataSaida = cursor.getLong(4);
        double orcamento = cursor.getDouble(5);

        item.put("id", id);

        if (tipoViagem == Constantes.VIAGEM_LAZER) {
            item.put("imagem", R.drawable.lazer);
        } else {
            item.put("imagem", R.drawable.negocios);
        }

        item.put("destino", destino);

        Date dataChegadaDate = new Date(dataChegada);
        Date dataSaidaDate = new Date(dataSaida);

        String periodo = dateFormat.format(dataChegadaDate) +
            " a " + dateFormat.format(dataSaidaDate);

        item.put("data", periodo);

        double totalGasto = calcularTotalGasto(db, id);

        item.put("total", "Gasto total R$ " + totalGasto);

        double alerta = orcamento * valorLimite / 100;
        Double [] valores =
```

```
            new Double[] { orcamento, alerta, totalGasto };
        item.put("barraProgresso", valores);

        viagens.add(item);

        cursor.moveToNext();
    }
    cursor.close();

    return viagens;
}
```

Para cada viagem da lista também deve ser apresentado o valor total gasto naquela viagem. Faremos essa implementação no método `calcularTotalGasto` , que realizará uma consulta que soma o valor dos gastos realizados com o `id` informado como parâmetro. O código desse método ficará da seguinte maneira:

```
private double calcularTotalGasto(SQLiteDatabase db,
        String id) {
    Cursor cursor = db.rawQuery(
        "SELECT SUM(valor) FROM gasto WHERE viagem_id = ?",
        new String[]{ id }
    );
    cursor.moveToFirst();
    double total = cursor.getDouble(0);
    cursor.close();
    return total;
}
```

O segundo parâmetro do método `rawQuery` espera um `Array` de `String` , e serve para informar os valores que serão usados na cláusula `WHERE` da consulta. Neste caso, queremos restringir a soma dos valores gastos apenas para a viagem, cujo `id` for igual ao informado.

Pronto! Agora temos nossa lista de viagens totalmente dinâmica, obtendo as informações diretamente do banco de dados.

Existem ainda duas outras formas de realizar consultas em um banco SQLite no Android. Uma delas é por meio dos métodos `query` , disponíveis na classe `SQLDatabase` . Com ela, devemos

informar cada trecho da nossa consulta através de um parâmetro; ou seja, precisa-se informar os campos devolvidos no `select`, no `groupBy`, no `having` e assim por diante. Veja a nossa primeira consulta reescrita utilizando essa forma:

```
String tabela = "viagem";
String[] colunas = new String[]{"_id", "tipo_viagem", "destino",
                                "data_chegada","data_saida",
                                "orcamento"};
String selecao = null;
String[] selecaoArgs = null;
String groupBy = null;
String having = null;
String orderBy = null;
Cursor cursor = db.query(tabela, colunas, selecao, selecaoArgs,
                         groupBy, having, orderBy);
```

A outra maneira de realizar consultas é usando um `SQLiteQueryBuilder`. Por meio dessa classe, é possível construir programaticamente consultas complexas, incluindo várias tabelas. A seguir, veja a nossa consulta construída utilizando essa abordagem:

```
SQLiteQueryBuilder builder = new SQLiteQueryBuilder();

builder.setTables("viagem");

Cursor cursor = builder.query(db, colunas, selecao,
                              selecaoArgs, groupBy,
                              having, orderBy);
```

4.4 ATUALIZAÇÃO DE VIAGENS E O UPDATE NO SQLITE

A próxima implementação que faremos em nosso aplicativo é a atualização das informações de uma determinada viagem. Quando a lista de viagens é apresentada e o usuário seleciona alguma delas, um `AlertDialog` com opções é exibido, sendo que uma delas diz respeito à edição da viagem selecionada. Implementaremos essa funcionalidade agora.

A ideia é que, quando o usuário selecionar a opção "editar", o `id` da viagem selecionada seja recuperado do mapa e colocado como um *extra* na `intent` que abrirá a `ViagemActivity` . Depois, no `onCreate` da `ViagemActivity` , recuperaremos esse valor para saber se estamos criando uma nova viagem ou editando uma existente.

No método `salvarViagem` , verificaremos este `id` para executar uma operação de `insert` ou de `update` . Veja o código do método `onClick` da classe `ViagemListActivity` para tratar a edição:

```
@Override
public void onClick(DialogInterface dialog, int item) {
    Intent intent;
    String id =
        (String) viagens.get(viagemSelecionada).get("id");

    switch (item) {
    case 0: // editar viagem
        intent = new Intent(this, ViagemActivity.class);
        intent.putExtra(Constantes.VIAGEM_ID, id);
        startActivity(intent);
        break;

    // códigos existentes
    ...
    }
}
```

Primeiro, recuperamos o `id` da viagem que o usuário deseja editar e o colocamos como informação extra na `intent` , que abrirá a `ViagemActivity` . Agora no método `onCreate` da `ViagemActivity` , obtemos esse valor e, caso seja válido, carregaremos as informações da viagem com o `id` informado, a partir do banco de dados, para exibir ao usuário. As alterações de código são as seguintes:

```
// demais atributos
private String id;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    //códigos existentes
    ...
    id = getIntent().getStringExtra(Constantes.VIAGEM_ID);

    if(id != null){
        prepararEdicao();
    }
}
```

No método `prepararEdicao`, buscaremos a viagem com o `id` informado e atribuiremos os valores obtidos aos widgets da tela.

```
private void prepararEdicao() {
    SQLiteDatabase db = helper.getReadableDatabase();

    Cursor cursor =
        db.rawQuery(
            "SELECT tipo_viagem, destino, data_chegada, " +
            "data_saida, quantidade_pessoas, orcamento " +
            "FROM viagem WHERE _id = ?", new String[]{ id }
        );

    cursor.moveToFirst();

    SimpleDateFormat dateFormat =
        new SimpleDateFormat("dd/MM/yyyy");

    if(cursor.getInt(0) == Constantes.VIAGEM_LAZER){
        radioGroup.check(R.id.lazer);
    } else {
        radioGroup.check(R.id.negocios);
    }

    destino.setText(cursor.getString(1));
    dataChegada = new Date(cursor.getLong(2));
    dataSaida = new Date(cursor.getLong(3));
    dataChegadaButton.setText(dateFormat.format(dataChegada));
    dataSaidaButton.setText(dateFormat.format(dataSaida));
    quantidadePessoas.setText(cursor.getString(4));
    orcamento.setText(cursor.getString(5));
    cursor.close();
}
```

O último passo é alterar o método `salvarViagem` para decidir se vamos realizar uma operação de `insert` ou de `update`, com

base no atributo `id` recuperado da `intent`. Se o `id` for igual a `null`, nada será informado na `intent` e trata-se de um novo registro de viagem. Caso contrário, usaremos o método `update`, que tem uma assinatura similar a do `insert`, recebendo dois parâmetros a mais para indicar os critérios de restrição. Informaremos nesses parâmetros que a atualização deve ser realizada para o registro com `_id` igual ao informado. As alterações no código do `salvarViagem` ficarão assim:

```
public void salvarViagem(View view) {
// códigos existentes
...

if(id == null){
    resultado = db.insert("viagem", null, values);
} else {
    resultado = db.update("viagem", values, "_id = ?",
                          new String[]{ id });
}

...
// códigos existentes
```

A operação de `update` retorna a quantidade de registros afetados pelo comando, e podemos utilizar essa informação para saber se a atualização deu certo. É isso, mais uma funcionalidade pronta! Já podemos testar a atualização de uma viagem existente. Agora só nos resta poder excluir as informações que não queremos mais.

4.5 COMO APAGAR UMA VIAGEM COM O SQLITE E O ANDROID

O que precisamos agora é implementar a exclusão de uma viagem e de seus respectivos gastos, quando a opção "Remover" for escolhida na `ViagemListActivity` e também a partir da mesma opção existente no menu da `ViagemActivity`. Assim como nas operações de `insert` e `update`, para a exclusão de registro, temos

o método `delete` disponível no `SQLiteDatabase`.

No `onClick` da `ViagemListActivity` incluiremos a chamada para um método que executará a exclusão dos registros. O método de exclusão só poderá ser chamado se o usuário confirmar a operação. Vamos alterar o código para que chame o método adequado.

```
@Override
public void onClick(DialogInterface dialog, int item) {
// códigos existentes
...
switch (item) {
...
case DialogInterface.BUTTON_POSITIVE: // exclusão
    viagens.remove(viagemSelecionada);
    removerViagem(id);
    getListView().invalidateViews();
    break;
...
}
```

O `id` da viagem a ser excluída será passado como parâmetro para o método `removerViagem`, no qual primeiramente excluímos todos os gastos associados e depois removemos a viagem propriamente dita.

```
private void removerViagem(String id) {
    SQLiteDatabase db = helper.getWritableDatabase();
    String where []  = new String[]{ id };
    db.delete("gasto", "viagem_id = ?", where);
    db.delete("viagem", "_id = ?", where);
}
```

Pronto, agora já conseguimos realizar as principais operações de banco de dados com o SQLite e o Android!

4.6 DICAS E BOAS PRÁTICAS AO TRABALHAR COM BANCO DE DADOS NO ANDROID

Para ser mais didático e também focar nas operações com o

banco de dados, não tivemos nenhuma preocupação em escrever um código bom e fácil de manter. No entanto, uma vez que já aprendemos a utilizar o SQLite e compreender como ele funciona, chegou o momento de aprimorar o código da nossa aplicação.

Basicamente, existem três problemas nos códigos que escrevemos. O primeiro deles é que não estamos usando nenhuma classe para representar o domínio da nossa aplicação. Poderíamos criar as classes `Viagem` e `Gasto` para representar esses objetos, obtendo, assim, um código mais organizado e de fácil manipulação; sem contar, é claro, com os demais benefícios do *design* orientado a objetos.

Como parte do processo de refatoração, podemos criar essas duas novas classes em um novo pacote chamado `br.com.casadocodigo.boaviagem.domain`. Veja a de `Viagem`:

```
public class Viagem {
    private Long id;
    private String destino;
    private Integer tipoViagem;
    private Date dataChegada;
    private Date dataSaida;
    private Double orcamento;
    private Integer quantidadePessoas;

    public Viagem(){}

    public Viagem(Long id, String destino, Integer tipoViagem,
            Date dataChegada, Date dataSaida, Double orcamento,
            Integer quantidadePessoas) {
        this.id = id;
        this.destino = destino;
        this.tipoViagem = tipoViagem;
        this.dataChegada = dataChegada;
        this.dataSaida = dataSaida;
        this.orcamento = orcamento;
        this.quantidadePessoas = quantidadePessoas;
    }

    // getters e setters
}
```

E, agora, a classe `Gasto` :

```
public class Gasto {

    private Long id;
    private Date data;
    private String categoria;
    private String descricao;
    private Double valor;
    private String local;
    private Integer viagemId;

    public Gasto(){}

    public Gasto(Long id, Date data, String categoria,
        String descricao, Double valor, String local,
        Integer viagemId) {

        this.id = id;
        this.data = data;
        this.categoria = categoria;
        this.descricao = descricao;
        this.valor = valor;
        this.local = local;
        this.viagemId = viagemId;
    }

    //getters e setters
}
```

Outro problema é que estamos manipulando diretamente diversas `String` que representam as tabelas e suas respectivas colunas. Se alguma delas mudar, teremos que varrer o código e alterar cada ocorrência dela.

Além disso, os códigos que se referem ao acesso aos dados estão misturados com códigos da `Activity` , que deveriam essencialmente tratar apenas da interação com o usuário. Novamente, qualquer alteração na estrutura do banco de dados influenciaria todo o restante do código da aplicação. Precisamos separar as responsabilidades.

Resolveremos esses problemas utilizando um objeto de acesso a

dados, o *Data Access Object* (DAO), que é um padrão para implementar a separação da lógica de negócio das regras de acesso a banco de dados. Podemos criar a classe `BoaViagemDAO` no pacote `br.com.casadocodigo.boaviagem.dao` que, inicialmente, terá o seguinte código:

```
public class BoaViagemDAO {

    private DatabaseHelper helper;
    private SQLiteDatabase db;

    public BoaViagemDAO(Context context){
        helper = new DatabaseHelper(context);
    }

    private SQLiteDatabase getDb() {
        if (db == null) {
            db = helper.getWritableDatabase();
        }
        return db;
    }

    public void close(){
        helper.close();
    }
}
```

O BoaViagemDAO define um construtor que recebe o contexto da aplicação e instancia um `DatabaseHelper` . Também definimos um método que retorna uma instância de `SQLiteDatabase` , criando-a se necessário. Usaremos sempre esse método para obter uma instância de `SQLiteDatabase` e, a partir dela, executar as operações com o banco de dados.

Você pode estar se perguntando por que fizemos isso, em vez de inicializar logo a variável `db` no construtor. O motivo é que o Android executa as operações de criação e atualização do banco apenas quando solicitamos uma instância de `SQLiteDatabase` , por meio do método `getWritableDatabase` ou do `getReadableDatabase` .

Como essas operações podem ser demoradas, recomenda-se não invocar esses métodos em construtores e métodos de inicialização, como no `onCreate` de uma `Activity` . Portanto, devemos postergar sua invocação até que realmente seja necessário executar uma operação com o banco de dados. No nosso DAO, também disponibilizamos o método `close` , que invoca o método de mesmo nome do `DatabaseHelper` para fechar o banco de dados aberto.

Antes de implementar os métodos de acesso a dados que são pertinentes para a nossa aplicação, vamos incluir algumas constantes para representar as tabelas e colunas existentes no banco de dados. Como as instruções de criação do banco ficam na classe `DatabaseHelper` , podemos definir essas constantes lá. Veja como ficou:

```
public class DatabaseHelper extends SQLiteOpenHelper{

    //constantes já existentes

    public static class Viagem {
        public static final String TABELA = "viagem";
        public static final String _ID = "_id";
        public static final String DESTINO = "destino";
        public static final String DATA_CHEGADA =
            "data_chegada";
        public static final String DATA_SAIDA = "data_saida";
        public static final String ORCAMENTO = "orcamento";
        public static final String QUANTIDADE_PESSOAS =
            "quantidade_pessoas";
        public static final String TIPO_VIAGEM =
            "tipo_viagem";

        public static final String[] COLUNAS = new String[]{
            _ID, DESTINO, DATA_CHEGADA, DATA_SAIDA,
            TIPO_VIAGEM, ORCAMENTO, QUANTIDADE_PESSOAS };
    }

    public static class Gasto{
        public static final String TABELA = "gasto";
        public static final String _ID = "_id";
        public static final String VIAGEM_ID = "viagem_id";
```

```
        public static final String CATEGORIA = "categoria";
        public static final String DATA = "data";
        public static final String DESCRICAO = "descricao";
        public static final String VALOR = "valor";
        public static final String LOCAL = "local";

        public static final String[] COLUNAS = new String[]{
            _ID, VIAGEM_ID, CATEGORIA, DATA, DESCRICAO,
            VALOR, LOCAL
        };
    }
    // demais códigos existentes
}
```

O `BoaViagemDAO` terá métodos que fazem exatamente o que já implementamos diretamente na `Activity`, com a diferença de que, em vez de retornar objetos do tipo `Cursor`, retornará listas de objetos de domínio e receberá objetos do tipo `Viagem` e `Gasto` para serem inseridos. Além disso, todas as referências às tabelas e colunas serão feitas usando as constantes definidas anteriormente. Um exemplo de método que terá no DAO é o `listarViagens`.

```
public List<Viagem> listarViagens(){
    Cursor cursor = getDb().query(DatabaseHelper.Viagem.TABELA,
                              DatabaseHelper.Viagem.COLUNAS,
                              null, null, null, null, null);
    List<Viagem> viagens = new ArrayList<Viagem>();
    while(cursor.moveToNext()){
        Viagem viagem = criarViagem(cursor);
        viagens.add(viagem);
    }
    cursor.close();
    return viagens;
}
```

Você pode consultar o código completo desse DAO no repositório do projeto do livro, disponível em https://github.com/joaobmonteiro/livro-android.

4.7 CONCLUSÃO

Neste capítulo, apresentamos os passos necessários para criar

um banco de dados SQLite no Android e também aprendemos como realizar as principais operações, como `insert`, `update`, `delete` e `query`.

Também vimos como executar instruções SQL arbitrárias com o `execSQL` e também consultas com o `rawQuery`. Além disso, utilizamos o padrão DAO para organizar melhor nosso código e fizemos algumas refatorações para melhorá-lo.

As funcionalidades implementadas são para a persistência das informações de viagens. Agora, fica como dever de casa implementar as mesmas funcionalidades para os gastos, já utilizando o DAO apresentado. Caso tenha dúvidas ou queira comparar sua implementação, consulte o código-fonte da aplicação, disponibilizado em http://github.com/joaobmonteiro/livro-android.

CAPÍTULO 5

COMPARTILHE DADOS ENTRE APLICAÇÕES COM CONTENT PROVIDERS

Os provedores de conteúdo (*content providers*) são componentes da plataforma Android usados para o compartilhamento de dados entre aplicações, que podem estar armazenados em um banco de dados SQLite local, em arquivos mantidos no próprio dispositivo ou mesmo na web.

O *content provider* responsabiliza-se por oferecer uma forma simples e segura para acessar e modificar esses dados, independentemente de onde estejam armazenados.

Geralmente, os provedores de conteúdo disponibilizam algum tipo de interface gráfica para manipular os dados, como é o caso do *provider* de contatos do telefone.

Neste capítulo, veremos como utilizar os provedores de conteúdo e também criaremos um *content provider* para compartilhar nossas informações de viagens, armazenadas no aplicativo BoaViagem.

5.1 COMO FUNCIONA UM CONTENT PROVIDER

O *content provider* expõe os dados para as outras aplicações em uma estrutura tabular, semelhante às tabelas dos bancos de dados relacionais. Cada linha representa uma instância do dado obtido pelo *provider*, e as colunas representam as informações referentes àquela instância.

Para acessar um provedor de conteúdo, utilizamos um objeto do tipo `ContentResolver` que se comunica com o `ContentProvider` e, este, por sua vez, recebe a solicitação, executa a ação desejada e retorna os resultados obtidos. As operações possíveis de um provedor de conteúdo são as de criar, recuperar, atualizar e remover dados.

Existem três elementos importantes em um `ContentProvider` : suas `Uris` , as colunas existentes e as permissões necessárias para acessá-lo. Uma `Uri` de conteúdo é uma `String` formada por um nome simbólico que identifica o provedor (a autoridade) por um caminho (o *path*) que indica em qual tabela os dados estão. Geralmente, os `ContentProviders` disponibilizam, em forma de constantes, uma `Uri` diferente para cada tabela que ele expõe.

Veja um exemplo de `Uri` , onde `com.android.contacts` é a autoridade do *provider* e `contacts` é o *path:*

```
content://com.android.contacts/contacts
```

Essa `Uri` , além de indicar de onde os dados devem ser obtidos, também pode conter informações sobre o dado propriamente dito. Por exemplo, um `ContentProvider` pode definir que alguns segmentos da `Uri` são, na verdade, parâmetros da operação que será executada. Isto é bastante comum em `Uris` utilizadas para recuperar um dado a partir de um `id` , ou para executar uma consulta informando um filtro que seja aplicado a várias colunas. Veremos exemplos assim mais adiante.

Uma vez determinada qual é a `Uri` de acesso, é necessário

saber em quais colunas as informações estão, para que se possam obter os dados. Os provedores de conteúdo disponibilizam o nome das colunas existentes também na forma de constantes. O provedor de contatos, por exemplo, agrupa todas as suas informações de `Uris` e colunas suportadas, na classe `ContactsContract`.

O `ContentProvider` também define permissões para que possa ser utilizado. Isso é especialmente importante, pois geralmente os provedores de conteúdo são usados para permitir que outras aplicações acessem e alterem dados de outra. Para utilizarmos um `ContentProvider` em nossa aplicação, precisamos incluir as permissões adequadas.

Uma aplicação que pretende, por exemplo, ler e alterar dados de contatos deve declarar as seguintes permissões no `AndroidManifest.xml`.

```
<uses-permission
    android:name="android.permission.READ_CONTACTS" />
<uses-permission
    android:name="android.permission.WRITE_CONTACTS" />
```

Agora que já temos permissões adequadas e conhecemos a `Uri` e as colunas que compõem a informação desejada, precisamos obter um `ContentResolver` para se comunicar com o `ContentProvider`. As classes que estendem `Context`, como `Activity` e `Service`, possuem um método `getContentResolver()`, que retornam uma instância desse tipo.

A partir do `ContentResolver`, podemos executar consultas, inclusão, alteração e exclusão de dados, se tivermos as permissões corretas e a operação for suportada pelo provedor. Os resultados recuperados de um `ContentProvider` são retornados em um `Cursor`.

5.2 ACESSE OS CONTATOS DO TELEFONE

O Android disponibiliza um `ContentProvider` bastante completo para compartilhar os dados dos contatos armazenados no telefone, cujas informações estão localizados em três tabelas principais.

Na tabela `ContactsContract.Contacts`, temos as informações básicas de uma determinada pessoa como nome (coluna `DISPLAY_NAME`), se ela possui telefone (coluna `HAS_PHONE_NUMBER`) e um `_ID` que pode ser usado para recuperar seus dados nas demais tabelas. Ela funciona como um agrupador já que uma pessoa pode ter diversos contatos.

A tabela `ContactsContract.RawContacts` relaciona os contatos de uma determinada pessoa, que podem ter vindo de serviços de sincronização diferentes (Google, Twitter ou outros). Nessa tabela, temos duas informações importantes, que são o nome (`ACCOUNT_NAME`) e o tipo (`ACCOUNT_TYPE`) da conta do serviço de sincronização.

Os dados dos contatos são efetivamente armazenados na tabela `ContactsContract.Data`. As demais tabelas existentes são auxiliares e ajudam na pesquisa.

Para executar uma consulta em um `ContentProvider`, devemos fornecer uma `Uri` que indique de onde os dados serão obtidos, uma projeção que representa quais informações devem ser recuperadas, os critérios de filtro e também a ordenação desejada, de forma bastante semelhante a uma consulta no SQLite.

Para conseguirmos um `Cursor` com as informações da tabela `Contacts`, primeiro precisamos conseguir o `ContentResolver`, por meio do contexto, invocando o método `getContentResolver()`, e, em seguida, definimos a `Uri` que devemos usar para o *provider* de contatos.

```
ContentResolver contentResolver = getContentResolver();
```

```
Uri uri = Contacts.CONTENT_URI;
```

O próximo passo é declararmos um `array` que representa as colunas que devem ser retornadas. Neste caso, escolhemos apenas o nome de exibição do contato. Não definimos nenhum critério de filtro e nem de ordenação.

```
// Recupera o ContentResolver e define a URI

String[] projection = new String[] { Contacts.DISPLAY_NAME };
String selection = null;
String[] selectionArgs = null;
String sortOrder = null;
```

Por último, invocamos o método que executa a consulta. Como resultado temos um `Cursor` .

```
// Recupera o ContentResolver e define a URI
// Monta as informações para a consulta

Cursor cursor = contentResolver.query(uri, projection,
    selection, selectionArgs, sortOrder);
```

Outra operação comum é realizar uma busca parcial, com base no nome do contato. Para esse tipo de consulta existe uma `Uri` específica, que realiza a busca parcial nos campos que identificam o contato, tais como: nome, sobrenome, apelido etc. Nesse caso, utilizaremos uma `Uri` de filtro, na qual adicionamos o critério desejado ao seu final, como sendo o seu último segmento.

Para conseguir filtrar todos os contatos que possuem a letra "A" em alguma parte do nome e ordenar os resultados de forma ascendente, vamos incluir o critério de filtro na `ContactsContract.Contacts.CONTENT_FILTER_URI` , usando o método `Uri.withAppendedPath` para construir a `Uri` necessária.

```
ContentResolver contentResolver = getContentResolver();
Uri uri =
    Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI, "A");
```

O resultado dessa operação é uma `Uri` com o seguinte

conteúdo: `content://com.android.contacts/contacts/filter/A` .

Repare que, nesse caso, não precisamos utilizar os parâmetros `selection` e `selectionArgs` para executar a consulta, visto que o provedor de contatos já disponibiliza uma forma específica de fazê-la.

```
String[] projection = new String[] { Contacts.DISPLAY_NAME };
String selection = null;
String[] selectionArgs = null;
String sortOrder = Contacts.DISPLAY_NAME + " ASC";

Cursor cursor = contentResolver.query(uri, projection,
        selection, selectionArgs, sortOrder);
```

Pronto, já temos a consulta pronta. Por fim, vale ressaltar que é obrigatório adicionar o critério de filtro na `Uri` e, caso ele não seja informado, este `ContentProvider` lançará uma exceção.

A fim de facilitar as consultas, o provedor de contatos disponibiliza entidades, como `Phone` e `Email` , que podem ser entendidas como tabelas geradas a partir de *joins* entre as três tabelas principais. No próximo código, usaremos a entidade `Phone` para recuperar os telefones do contato que possui o `_ID` igual a 10:

```
ContentResolver contentResolver = getContentResolver();
Uri uri = Phone.CONTENT_URI;

String[] projection = new String[] { Phone.NUMBER,
                                     Phone.TYPE };
String selection = Phone.CONTACT_ID + " = ?";
String[] selectionArgs = new String[]{ "10" };
String sortOrder = null;

Cursor cursor = contentResolver.query(uri, projection,
        selection, selectionArgs, sortOrder);
```

O código é basicamente o mesmo utilizado anteriormente para listar os contatos, modificamos apenas a `Uri` e as colunas que agora pertencem à entidade `Phone` . Tanto a entidade `Phone`

quanto a `Email` possuem uma coluna `TYPE` que indica o seu respectivo tipo. Para saber mais sobre o `ContentProvider` de contatos, consulte a documentação.

5.3 CRIE UM CONTENTPROVIDER PARA O SEU APLICATIVO

O `ContentProvider` é um importante componente da plataforma Android que essencialmente permite o compartilhamento de dados entre as aplicações, disponibilizando operações de leitura e escrita. Por questões de segurança e por padrão, não é possível acessar qualquer tipo de dado que pertença a outra aplicação, seja ele um arquivo (de imagem, vídeo, texto etc.) ou um banco de dados SQLite. Não há necessidade de implementar um provedor de conteúdo se não for necessário compartilhar dados com outros aplicativos.

No entanto, pode haver necessidade de que nosso aplicativo possa transferir ou receber dados de outro. Já vimos que é possível passar alguns tipos de dados via `Intents`, mas, quando o volume de informações é maior e mais estruturado, isso não é suficiente. A saída existente é implementar um `ContentProvider` para tomar conta dessa comunicação entre aplicativos.

Como exemplo, suponha que precisamos compartilhar, seja por necessidade ou por diferencial, as informações de viagens e gastos com qualquer outra aplicação. Para isso, criaremos um provedor de conteúdo para o BoaViagem que consiste em estender a classe `ContentProvider` e implementar alguns de seus métodos. Além disso, definiremos as `Uris` e as permissões necessárias para acessar o *provider*.

Por questões de organização, vamos criar todas as classes relacionadas ao *provider* de conteúdo no pacote

br.com.casadocodigo.boaviagem.provider . Crie nele uma nova classe chamada de BoaViagemProvider e faça com que ela estenda ContentProvider . A seguir está o código da classe com os métodos ainda sem implementação. Discutiremos cada um deles na sequência.

```
public class BoaViagemProvider extends ContentProvider{

    @Override
    public boolean onCreate() {
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection,
            String selection, String[] selectionArgs,
            String sortOrder) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
    public int delete(Uri uri, String selection,
            String[] selectionArgs) {
        return 0;
    }

    @Override
    public int update(Uri uri, ContentValues values,
            String selection, String[] selectionArgs) {
        return 0;
    }

    @Override
    public String getType(Uri uri) {
        return null;
    }
}
```

O ContentProvider depende de uma Uri , que além de determinar a operação desejada, indica também sobre quais dados

ela deve ser executada.

No nosso caso, disponibilizaremos cinco `Uris` com objetivos distintos:

- Inserção ou pesquisa de viagens;
- Atualização ou remoção de viagens;
- Pesquisa de gastos de uma viagem;
- Inserção ou pesquisa de um gasto;
- Atualização ou remoção de um gasto.

Cada `Uri` deverá possuir um segmento que identifica o provedor, também conhecido como autoridade (*authority*), e o *path* que indicará a localização dos dados. Adicionalmente, ela pode ter um segmento que indica o `_ID` do dado desejado.

Sua estrutura é muito importante e deve ser escolhida com cuidado, pois é a partir dela que o provedor tomará as decisões sobre o que executar.

Para as `Uris` de viagens, teremos uma para inserir ou pesquisar viagens e a outra para atualizar ou remover um registro, sendo necessário, neste último caso, informar o `_ID` da viagem como parte dela. Faremos o mesmo para as `Uris` de gastos, além de disponibilizar uma especificamente para recuperar os de uma determinada viagem. A ideia é que elas sejam assim:

- `content://br.com.casadocodigo.boaviagem.provider/viagem` — inserir ou pesquisar viagens;

- `content://br.com.casadocodigo.boaviagem.provider/viagem/#` — atualizar ou remover viagem;

- `content://br.com.casadocodigo.boaviagem.provider/gasto/viagem/#` — pesquisar gastos de uma viagem;

- content://br.com.casadocodigo.boaviagem.provider/gasto — inserir ou pesquisar gastos;
- content://br.com.casadocodigo.boaviagem.provider/gasto/# — atualizar ou remover gasto.

O # indica que aquele segmento deve corresponder a uma sequência de caracteres numéricos de qualquer tamanho; ou seja, deve ser um número que representará o _ID do dado desejado.

Uma Uri também pode conter * , que indica que o segmento deve ser uma String de qualquer tamanho. Como as boas práticas recomendam, usaremos constantes para representá-las. Também é importante criar constantes que representam as colunas que serão recuperadas tanto para facilitar o uso por terceiros quanto para organizarmos internamente o código e, assim, evitarmos o uso explícito de String e Integer .

Vale salientar que essas colunas não são necessariamente as mesmas do banco SQLite da aplicação. Podemos expor um conjunto diferente de dados que, inclusive, pode não refletir a estrutura de tabelas existentes no banco de dados. Portanto, lembre-se de que, apesar da similaridade, o *path* e as colunas do *provider* não são a mesma coisa que as tabelas e colunas do banco de dados.

Crie uma classe BoaViagemContract que definirá as colunas e Uris do nosso ContentProvider , por meio de constantes. Ela terá o seguinte código:

```
public final class BoaViagemContract {
    public static final String AUTHORITY =
            "br.com.casadocodigo.boaviagem.provider";
    public static final Uri AUTHORITY_URI =
            Uri.parse("content://" + AUTHORITY);
    public static final String VIAGEM_PATH = "viagem";
    public static final String GASTO_PATH = "gasto";

    public static final class Viagem{
```

```
        public static final Uri CONTENT_URI =
            Uri.withAppendedPath(AUTHORITY_URI, VIAGEM_PATH);
        public static final String _ID = "_id";
        public static final String DESTINO = "destino";
        public static final String DATA_CHEGADA =
            "data_chegada";
        public static final String DATA_SAIDA = "data_saida";
        public static final String ORCAMENTO = "orcamento";
        public static final String QUANTIDADE_PESSOAS =
                "quantidade_pessoas";
    }

    public static final class Gasto{
        public static final Uri CONTENT_URI =
            Uri.withAppendedPath(AUTHORITY_URI, GASTO_PATH);
        public static final String _ID = "_id";
        public static final String VIAGEM_ID = "viagem_id";
        public static final String CATEGORIA = "categoria";
        public static final String DATA = "data";
        public static final String DESCRICAO = "descricao";
        public static final String LOCAL = "local";
    }
}
```

Para criar as `Uris`, utilizamos dois métodos utilitários: o `Uri.parse`, que cria uma `Uri` válida a partir de uma `String`; e o `Uri.withAppendedPath`, que é usado para incluir um novo segmento a uma `Uri` existente.

Quando alguma operação for solicitada para o nosso `ContentProvider`, precisaremos a partir da `Uri` determinar o que deve ser feito. Para isso, utilizaremos a classe utilitária `UriMatcher` que faz a comparação da `Uri` informada pelo usuário com as definidas, retornando um valor previamente definido indicando qual foi a `Uri` em que houve o *matching*, como a seguir:

```
int VIAGENS = 1;

UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
uriMatcher.addURI(AUTHORITY, VIAGEM_PATH, VIAGENS);

if(uriMatcher.match(uri) == VIAGENS){
    // operação de listar viagens
```

```
}
```

Nesse trecho de código, criamos a `UriMatcher` e adicionamos a `Uri` que corresponde à operação de pesquisar viagens. O valor `VIAGENS` será usado para comparar o retorno do método `UriMatcher.match` e, caso sejam iguais, quer dizer que a `Uri` informada é a de pesquisa de viagens.

É necessário adicionar ao `UriMatcher` cada `Uri` que será utilizada pelo provedor. Portanto, na classe `BoaViagemProvider` teremos o código para isso:

```
import static br.com.casadocodigo.boaviagem.BoaViagemContract.*;

public class BoaViagemProvider extends ContentProvider {

    private static final int VIAGENS = 1;
    private static final int VIAGEM_ID = 2;
    private static final int GASTOS = 3;
    private static final int GASTO_ID = 4;
    private static final int GASTOS_VIAGEM_ID = 5;

    private static final UriMatcher uriMatcher =
        new UriMatcher(UriMatcher.NO_MATCH);

    static{
        uriMatcher.addURI(AUTHORITY, VIAGEM_PATH, VIAGENS);

        uriMatcher.addURI(AUTHORITY,
            VIAGEM_PATH + "/#", VIAGEM_ID);

        uriMatcher.addURI(AUTHORITY, GASTO_PATH, GASTOS);

        uriMatcher.addURI(AUTHORITY,
            GASTO_PATH + "/#", GASTO_ID);

        uriMatcher.addURI(AUTHORITY,
            GASTO_PATH + "/"+ VIAGEM_PATH + "/#",
            GASTOS_VIAGEM_ID);
    }

    // demais códigos existentes
}
```

Assim, adicionamos as cinco `Uris` que utilizaremos, e

poderemos posteriormente invocar o método `UriMatcher.match` para determinar qual operação deve ser realizada.

Já temos as definições e a estrutura do `ContentProvider` prontas e agora partiremos para a implementação dos seus métodos. Como nossos dados estão armazenados em um banco `SQLite`, usaremos a nossa classe `DatabaseHelper` para acessá-lo. A sua instanciação será feita no método `onCreate` do `BoaViagemProvider`.

```
private DatabaseHelper helper;

@Override
public boolean onCreate() {
    helper = new DatabaseHelper(getContext());
    return true;
}
```

O próximo método a ser implementado é o `query`, para realizar consultas no provedor de conteúdo. A partir da `Uri` informada como parâmetro, determinaremos qual é a consulta que deve ser realizada. Começaremos pelas consultas de viagens e vamos precisar de uma instância de `SQLiteDatabase`, que conseguimos por meio do `DatabaseHelper` que acabamos de instanciar no método `onCreate`.

```
@Override
public Cursor query(Uri uri, String[] projection,
        String selection, String[] selectionArgs,
        String sortOrder) {

    SQLiteDatabase database = helper.getReadableDatabase();

    // as consultas virão aqui
}
```

Agora temos que fazer a comparação da `Uri` informada com aquelas carregadas previamente no `UriMatcher`. Caso a informada seja a de `VIAGENS`, executamos uma consulta na tabela `viagem`.

```
    switch (uriMatcher.match(uri)) {
```

```
        case VIAGENS:
            return database.query(VIAGEM_PATH, projection,
            selection, selectionArgs, null, null, sortOrder);
    }
```

Aqui, propositadamente fizemos o *path* coincidir com o nome da tabela a que ele se refere. Repare também que os parâmetros recebidos pelo método `ContentProvider.query` são praticamente os mesmos recebidos pelo `SQLiteDatabase.query` . Então, simplesmente os repassamos para que a consulta seja executada. Lembre-se de que, em alguns casos, pode ser necessário validar ou checar os parâmetros recebidos, para garantir a execução correta da operação.

Caso a `Uri` for `VIAGEM_ID` , a nossa consulta deverá ter uma cláusula `where` para restringir a consulta com base no `_ID` informado. Para isso, recuperamos o último segmento da `Uri` , que representa o `_ID` do registro, utilizando o método `uri.getLastPathSegment` , e executamos a consulta desejada.

Por fim, no caso da `Uri` informada não coincidir com nenhuma das definidas no `UriMatcher` , lançamos uma exceção.

```
case VIAGEM_ID:
    selection = Viagem._ID + " = ?";
    selectionArgs = new String[] {uri.getLastPathSegment()};
    return database.query(VIAGEM_PATH, projection,
        selection, selectionArgs, null, null, sortOrder);
default:
    throw new IllegalArgumentException("Uri desconhecida");
```

Agora podemos fazer as consultas para os gastos, que funcionarão de forma similar às de viagens. Vamos usar as constantes `GASTOS` e `GASTO_ID` :

```
case GASTOS:
    return database.query(GASTO_PATH, projection,
        selection, selectionArgs, null, null, sortOrder);

case GASTO_ID:
```

```
    selection = Gasto._ID + " = ?";
    selectionArgs = new String[] {uri.getLastPathSegment()};
    return database.query(GASTO_PATH, projection,
        selection, selectionArgs, null, null, sortOrder);
```

Utilize como restrição a `VIAGEM_ID` . Veja como ficou o código completo do método `query` :

```
case GASTOS_VIAGEM_ID:

    selection = Gasto.VIAGEM_ID + " = ?";
    selectionArgs = new String[] {uri.getLastPathSegment()};
    return database.query(GASTO_PATH, projection,
        selection, selectionArgs, null, null, sortOrder);
```

A implementação do método `insert` é bastante simples, pois já recebemos como parâmetro um `ContentValues` que contém os dados que o usuário deseja inserir, e também não fazemos nenhuma restrição. Ao contrário do método `insert` do `SQLiteDatabase` , que retorna o `_ID` do registro inserido; o do `ContentProvider` retorna uma `Uri` que o representa. Dessa forma, após realizarmos a inserção, retornaremos uma `Uri` contendo o `_ID` do novo registro.

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    SQLiteDatabase database = helper.getWritableDatabase();
    long id;

    switch (uriMatcher.match(uri)) {

    case VIAGENS:
        id = database.insert(VIAGEM_PATH, null, values);
        return Uri.withAppendedPath(Viagem.CONTENT_URI,
                                    String.valueOf(id));

    case GASTOS:
        id = database.insert(GASTO_PATH, null, values);
        return Uri.withAppendedPath(Gasto.CONTENT_URI,
                                    String.valueOf(id));
    default:
        throw new IllegalArgumentException("Uri desconhecida");
    }
```

```
}
```

A implementação do método `update` é semelhante à realizada no `insert` , já que também temos o `ContentValues` como parâmetro. A diferença é que recebemos dois parâmetros a mais que representam a cláusula `where` da instrução SQL de `UPDATE` ; o primeiro é uma `string` com os campos e o segundo é um `array` com os respectivos valores. O método `delete` também recebe esses dois, além da `Uri` .

Agora precisamos adicionar suporte à alteração e exclusão de informações, de acordo com o `_ID` . Usaremos as `Uris` que contenham o `_ID` do dado, além de substituir os parâmetros recebidos para incluir essa restrição. Para a exclusão tanto do gasto quanto da viagem, faremos uso do método `delete` do `SQLiteDatabase` .

```
@Override
public int delete(Uri uri, String selection,
                    String[] selectionArgs) {

    SQLiteDatabase database = helper.getWritableDatabase();

    switch (uriMatcher.match(uri)) {

    case VIAGEM_ID:

        selection = Viagem._ID + " = ?";
        selectionArgs =
            new String[] {uri.getLastPathSegment()};
        return database.delete(VIAGEM_PATH,
                                selection, selectionArgs);

    case GASTO_ID:

        selection = Gasto._ID + " = ?";
        selectionArgs =
            new String[] {uri.getLastPathSegment()};
        return database.delete(GASTO_PATH,
                                selection, selectionArgs);

    default:
        throw new IllegalArgumentException("Uri desconhecida");
```

```
    }
}
```

E para a alteração, temos que chamar o método `update` do `SQLiteDatabase`.

```
@Override
public int update(Uri uri, ContentValues values,
                  String selection, String[] selectionArgs) {

    SQLiteDatabase database = helper.getWritableDatabase();

    switch (uriMatcher.match(uri)) {

    case VIAGEM_ID:

        selection = Viagem._ID + " = ?";
        selectionArgs = new String[] {uri.getLastPathSegment()};
        return database.update(VIAGEM_PATH, values,
                               selection, selectionArgs);

    case GASTO_ID:

        selection = Gasto._ID + " = ?";
        selectionArgs = new String[] {uri.getLastPathSegment()};
        return database.update(GASTO_PATH, values,
                               selection, selectionArgs);

    default:
        throw new IllegalArgumentException("Uri desconhecida");
    }
}
```

O último método do provedor de conteúdo que deve ser implementado é o `getType`. Ele deve retornar uma `String` no formato MIME que representa o tipo de dado retornado por uma determinada `Uri`. Como nosso *provider* lida com tabelas, é necessário retornar um formato MIME específico do Android (*vendor-specific MIME format*). Se nosso `ContentProvider` utilizasse arquivos, poderíamos usar os tipos MIME existentes, como por exemplo `image/jpeg`.

Quando se trata de tabelas, podemos retornar uma determinada

linha ou um conjunto. O Android disponibiliza, respectivamente, os tipos `vnd.android.cursor.item` e `vnd.android.cursor.dir` para esses casos. Além disso, também temos que identificar o *provider* e o tipo do dado retornado. Veja como fica a `String` que representa o tipo MIME das `Uris` de viagem:

- Uma única viagem:

```
"vnd.android.cursor.item/vnd.br.com.casadocodigo.
    boaviagem.provider/viagem"
```

- Uma lista de viagens:

```
"vnd.android.cursor.dir/vnd.br.com.casadocodigo.
    boaviagem.provider/viagem"
```

Esses tipos também devem constar na nossa classe de contrato `BoaViagemContract`. Logo, vamos incluir mais algumas constantes para representá-los:

```
public static final class Viagem {
    public static final String CONTENT_TYPE =
            "vnd.android.cursor.dir/" +
            "vnd.br.com.casadocodigo.boaviagem.provider/viagem";
    public static final String CONTENT_ITEM_TYPE =
            "vnd.android.cursor.item/" +
            "vnd.br.com.casadocodigo.boaviagem.provider/viagem";

    // demais constantes
}

public static final class Gasto {
    public static final String CONTENT_TYPE =
            "vnd.android.cursor.dir/"+
            "vnd.br.com.casadocodigo.boaviagem.provider/gasto";
    public static final String CONTENT_ITEM_TYPE =
            "vnd.android.cursor.item/" +
            "vnd.br.com.casadocodigo.boaviagem.provider/gasto";

    // demais constantes
}
```

A implementação do método `getType` retornará esses tipos, de acordo com a `Uri` informada. Assim, precisamos implementar o

`switch` que fará a devolução da `String` adequada, referente ao `CONTENT_TYPE`.

```
@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {

    case VIAGENS:
        return Viagem.CONTENT_TYPE;

    case VIAGEM_ID:
        return Viagem.CONTENT_ITEM_TYPE;

    case GASTOS:

    case GASTOS_VIAGEM_ID:
        return Gasto.CONTENT_TYPE;

    case GASTO_ID:
        return Gasto.CONTENT_ITEM_TYPE;

    default:
        throw new IllegalArgumentException("Uri desconhecida");
    }
}
```

No caso da `Uri` ser de `GASTOS` ou `GASTOS_VIAGEM_ID`, o tipo retornado é sempre uma lista de gastos; por isso, o `CONTENT_TYPE` deve ser o mesmo, ficando o `case` de `GASTOS` em branco. Agora que já temos todos os métodos implementados, só falta adicionar o `ContentProvider` no `AndroidManifest.xml` e ele já estará pronto para ser usado! Adicione um elemento `<provider>`, dessa forma:

```
<provider
    android:name=".provider.BoaViagemProvider"
    android:authorities=
        "br.com.casadocodigo.boaviagem.provider" >
</provider>
```

E pronto, acabamos de implementar nosso *provider*.

5.4 ADICIONE REGRAS DE PERMISSÃO AO

SEU CONTENTPROVIDER

Por padrão, o `ContentProvider` não possui nenhum tipo de permissão. Dessa forma, qualquer aplicativo pode acessar o provedor e realizar operações; o que pode ser algo indesejável, dependendo da circunstância. No entanto, é possível acrescentar permissões globais para a leitura e escrita, para apenas algumas tabelas, para alguns registros, ou também combinar todos esses tipos de permissões.

As permissões são definidas no manifesto juntamente com a declaração do provedor. Elas também precisam ter um nome único, por isso, utilizamos o nome do pacote em sua composição.

Para definir uma permissão global para escrita e leitura do nosso provedor, podemos usar o atributo `android:permission` do elemento `<provider>`, assim:

```
<provider
    android:name=".provider.BoaViagemProvider"
    android:authorities="br.com.casadocodigo.boaviagem.provider"
    android:permission=
        "br.com.casadocodigo.boaviagem.provider.permission.ALL">
</provider>
```

Para dividir as permissões de leitura e escrita, podemos usar o `android:readPermission` e o `android:writePermission`.

```
<provider
    android:name=".provider.BoaViagemProvider"
    android:authorities="br.com.casadocodigo.boaviagem.provider"
    android:readPermission=
        "br.com.casadocodigo.boaviagem.provider.permission.READ"
    android:writePermission=
"br.com.casadocodigo.boaviagem.provider.permission.WRITE">
</provider>
```

De forma mais específica, podemos utilizar o elemento `<path-permission>` para definir permissões para um determinado *path*, sendo possível, assim, restringir o acesso a tabelas e registros. Desse

modo, podemos restringir acesso às viagens:

```
<provider
    android:name=".provider.BoaViagemProvider"
    android:authorities=
        "br.com.casadocodigo.boaviagem.provider">
    <path-permission
        android:path="viagem"
        android:permission=
    "br.com.casadocodigo.boaviagem.provider.permission.ALL" />
</provider>
```

O aplicativo que deseja usar o `BoaViagemProvider` deverá incluir, em seu respectivo manifesto, as permissões corretas para conseguir acessá-lo.

5.5 CONCLUSÃO

Neste capítulo, exploramos mais um dos componentes da plataforma, o `ContentProvider` , cuja função principal é o compartilhamento de dados entre aplicações. Entendemos como os provedores de conteúdo funcionam, por meio de exemplos de uso do provedor de contatos. Por fim, criamos o nosso próprio `ContentProvider` para o aplicativo BoaViagem, com permissões específicas para escrita e leitura.

CAPÍTULO 6

INTEGRAÇÃO DE APLICAÇÕES ANDROID COM SERVIÇOS REST

Nossa aplicação atingiu um conjunto de funcionalidades razoável, com o qual podemos cadastrar e pesquisar viagens e gastos. Porém, algumas vezes, funcionalidades interessantes acabam indo além dos dados que temos disponíveis em nossa aplicação. Colocar uma mensagem direto nas redes sociais do usuário e descobrir o que as pessoas comentam sobre um determinado assunto são exemplos disso. Como fazemos para ter os dados dessas aplicações?

Algo que tem se tornado bastante comum é que as aplicações tanto web quanto *mobile* e até mesmo aplicações *desktop*, utilizem serviços remotos disponibilizados por outras aplicações. Os usuários esperam cada vez mais centralização e integração de dados, com o intuito de trazer acesso rápido e comodidade, obtendo as informações com a ponta dos dedos, literalmente.

Grandes empresas como Facebook, Twitter, Google e Yahoo! disponibilizam acesso remoto a seus serviços através de uma API, e podemos utilizá-los para enriquecer nossas aplicações.

Neste capítulo, você aprenderá como consumir serviços remotos em uma aplicação Android. Aproveitaremos também para aprender

mais sobre outros componentes da plataforma, os `Services` e os `BroadcastReceivers`.

TERMINOLOGIA

Usaremos muitos vezes o termo **serviço**, ao longo deste capítulo, como sinônimo de serviços remotos e *webservices*. Para evitar confusão, quando estivermos nos referindo ao componente da plataforma Android de mesmo nome, utilizaremos o termo *service*.

6.1 TRABALHE COM REST E JSON

Um tipo de *webservice* em especial tem se consolidado como padrão quando se trata de disponibilizar serviços na web. São os serviços do tipo REST, que possuem basicamente cinco premissas:

- Utilizar os métodos do protocolo HTTP para representar as operações que podem ser executadas pelo serviço;
- Expor as informações através de URLs representativas, similar a uma estrutura de diretórios;
- O serviço não deve armazenar estado entre requisições;
- Transmitir os dados em formato XML e/ou JSON;
- O uso de *hypermedia* para representar possíveis transições.

Como resultado, o que temos, então, é uma URL que, quando acessada usando o método HTTP correto e os parâmetros necessários, retorna dados em formato texto. De forma simplista, isso é um serviço REST. Esta simplicidade e facilidade permitem seu uso em praticamente qualquer tipo de plataforma, desde web até

mobile. Para saber mais sobre o assunto, confira o livro *REST: Construa API's inteligentes de maneira simples* da Casa do Código em http://www.casadocodigo.com.br/products/livro-rest.

Quando pedimos informações para um outro serviço, ele precisa nos transferir os dados que pedimos, para que possamos decidir o que fazer com ele em nossa aplicação. Para trafegar as informações, poderíamos usar tanto o conhecido formato XML como o JSON (*JavaScript Object Notation*), que é um formato simples utilizado para representar dados, voltado principalmente para a conversão de dados estruturados para a forma textual.

Esse formato é capaz de representar quatro tipos primitivos (números, `strings`, valores booleanos e `null`) e dois tipos estruturados (objetos e `arrays`). Considerando as informações de um livro, teríamos um JSON parecido com o seguinte:

```
{
    "Livro": {
        "titulo": "Introdução ao desenvolvimento Android",
        "editora": "Casa do Código",
        "autores": [
            "João Bosco O. Monteiro"
        ],
        "anoPublicacao": 2012
    }
}
```

Vamos levar em consideração um aplicativo para auxiliar o usuário a manter registros sobre os livros que ele leu e também dos que possui. Seria interessante que o usuário pudesse pesquisar e escolher o livro em vez de ter que cadastrar todas as suas informações, como: título, autor, editora etc. Não seria mais simples caso existisse um serviço já pronto que, através de uma URL, nos fornecesse os dados de um determinado livro, em um formato simples de processar, como o JSON?

Pois bem, esse serviço existe e chama-se Google Books API.

Você pode acessá-lo usando o seu próprio navegador, para ver como funciona um serviço REST e verificar os dados retornados em formato JSON. Para testar, acesse a seguinte URL: https://www.googleapis.com/books/v1/volumes?q=android.

O resultado obtido é uma lista de livros, contendo título, autores, data de publicação e várias outras informações.

```
{
 "kind": "books#volumes",
 "totalItems": 1253,
 "items": [
  {
   "kind": "books#volume",
   "id": "wH1gzgAACAAJ",
   "etag": "esMB5pqt+jI",
   "selfLink":
"https://www.googleapis.com/books/v1/volumes/wH1gzgAACAAJ",
   "volumeInfo": {
    "title": "Android in Action",
    "authors": [
     "W. Frank Ableson",
     "Robi Sen",
     "Chris King",
     "C. Enrique Ortiz"
    ],
    "publisher": "Manning Pubns Co",
    "publishedDate": "2011-09-28",
    "description": "Android is a free, open source...",
    "industryIdentifiers": [
     {
      "type": "ISBN_10",
      "identifier": "1617290505"
     },
     {
      "type": "ISBN_13",
      "identifier": "9781617290503"
     }
    ]
```

A Google Books API é bastante poderosa e utilizamos apenas uma das funcionalidades para exemplificar um serviço REST. Analisando esse JSON, você pode achar que será complicado trabalhar com esse formato. Porém, fique tranquilo, pois o Android

possui algumas classes como a `JSONObject` para facilitar o trabalho. Além disso, existem outras bibliotecas como a Gson (http://code.google.com/p/google-gson/), do próprio Google, e a Jackson (http://jackson.codehaus.org/), que trazem ainda mais facilidades para a manipulação de dados nesse formato.

Agora que já sabemos o básico sobre REST, que tal implementar um aplicativo que realiza consultas no Twitter e também notifica quando alguém nos menciona em algum *tweet*? Essa é uma situação interessante, pois será construída com implementações que são bastante comuns quando se trabalha com serviços REST.

Por exemplo, usaremos tarefas assíncronas (`AsyncTasks`) do Android para realizar requisições HTTP, criaremos `threads` e realizaremos execuções periódicas em um `Service` do Android. Além disso, utilizaremos um `BroadcastReceiver` para iniciar automaticamente nosso aplicativo, quando o dispositivo for iniciado.

6.2 CONHEÇA A TWITTER SEARCH API

Para trabalharmos com a API 1.1 do Twitter, precisamos obter credenciais de acesso para poder interagir com os serviços. Primeiro, devemos criar uma aplicação no Twitter para obter uma `consumer key` e um `consumer secret` . Acesse https://apps.twitter.com e crie uma nova aplicação, preenchendo os campos obrigatórios. Depois, navegue até a opção `API Keys` , e lá estarão listadas a `key` e o `secret` .

A partir dessas duas informações, faremos uma requisição para o Twitter que nos retornará um *token* de acesso para realizar chamadas aos serviços REST. Vamos ter que lidar com requisições HTTP a partir deste momento, e usaremos um pequeno framework para nos auxiliar nessa tarefa, o `http-request` .

Este é um projeto open source hospedado no GitHub bastante utilizado por desenvolvedores Android. Acesse o site https://github.com/kevinsawicki/http-request e copie o arquivo `HttpRequest.java` para a pasta de códigos-fonte do seu projeto. Você pode também copiar o arquivo `.jar` do `http-request`, disponível no repositório Maven Central, e colocá-lo na pasta `libs` do seu projeto.

Já podemos iniciar a implementação da nossa busca no Twitter! Crie um novo projeto com o nome de `TwitterSearch`. Após sua criação, não se esqueça de incluir a permissão para acesso à Internet no `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

A nossa *activity* principal terá um layout simples, com um `EditText` para o usuário informar os termos da pesquisa, um `Button` para disparar a consulta e uma `ListView` para apresentar os resultados no arquivo `main.xml`. A figura a seguir mostra como ficará a nossa aplicação.

Figura 6.1: Pesquisa no Twitter

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <EditText
            android:id="@+id/texto"
            android:layout_width="250dp"
            android:layout_height="wrap_content"
            android:inputType="text"
            android:singleLine="true" />
```

```
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="buscar"
            android:text="@string/buscar" />
    </LinearLayout>

    <ListView
        android:id="@+id/lista"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >
    </ListView>

</LinearLayout>
```

Precisamos também das mensagens, no arquivo `strings.xml` .

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">TwitterSearch</string>
    <string name="buscar">Buscar</string>
</resources>
```

Crie uma atividade chamada `TwitterSearchActivity` que, por enquanto, apenas inicializa as variáveis correspondentes aos *widgets* e define o método `buscar` , ficando assim:

```
public class TwitterSearchActivity extends Activity {

    private ListView lista;
    private EditText texto;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        lista = (ListView) findViewById(R.id.lista);
        texto = (EditText) findViewById(R.id.texto);
    }

    public void buscar(View v) {

    }
}
```

Para realizar a consulta no Twitter, faremos uma requisição HTTP. No entanto, não devemos (e nem podemos) acessar a rede a partir da `thread` que está rodando a *activity* (UI *thread*). Qualquer tipo de processamento ou operação que seja demorada não deve ser executada na UI `thread`, pois isto bloquearia a interface gráfica e poderia causar o erro de *Application Not Responding* (ANR).

Especificamente no caso de acesso à Internet, a partir da `thread` principal, o Android lança uma exceção e não permite a realização da operação.

Para essa situação, o Android disponibiliza, por meio da classe `AsyncTask`, uma forma simples de criar tarefas assíncronas que executam operações em *background* (em outra `thread`) e que podem publicar os resultados da operação na UI `thread`. Vamos usá-la para realizar a busca no Twitter, atualizar a nossa `ListView` e controlar um `ProgressDialog`. A `AsyncTask` é uma classe genérica e utiliza três tipos:

- `Params` — é o tipo dos parâmetros que são enviados para a execução da tarefa. Por exemplo, pode ser uma `String` que representa a URL da pesquisa que deve ser realizada.
- `Progress` — é o tipo que representa a unidade de progresso da tarefa. Pode ser o `Integer`, representando a porcentagem de progresso de uma operação de download ou ainda uma classe sua que contenha outros atributos que possam realizar esta indicação.
- `Result` — é o tipo de retorno da operação realizada. Pode ser um `array` com todos os *tweets* obtidos na consulta, por exemplo.

Quando uma `AsyncTask` é executada, ela percorre quatro

etapas (métodos):

1. `onPreExecute` — invocado na UI `thread`, antes da tarefa ser executada. Neste passo, geralmente preparamos a execução da operação, o que pode, por exemplo, incluir a exibição de um `ProgressDialog`.
2. `doInBackground` — é invocado em uma outra `thread` e é onde a operação deve ser implementada. Este método recebe os `Params` definidos pela `AsyncTask` e retorna os resultados da operação como sendo do tipo `Result`. Nele, podemos invocar o `publishProgess` para informar o andamento da operação.
3. `onProgressUpdate` — este método é invocado na UI `thread`, após o `publishProgress` ser executado. Aqui, podemos utilizar o valor (do tipo `Progress`) informado para fazer possíveis atualizações de tela, como por exemplo, atualizar o progresso de um `ProgressBar`.
4. `onPostExecute` — também invocado na UI `thread`, ele recebe o `Result` como parâmetro e faz as atualizações de tela necessárias, como fechar o `ProgressDialog` e atualizar uma `ListView` com os dados obtidos do método `doInBackground`.

O único método que devemos obrigatoriamente implementar quando estendemos a classe `AsyncTask` é o `doInBackground`.

Dentro da `TwitterSearchActivity`, vamos criar duas classes privadas que estendem de `AsyncTask`: uma para obter o token de acesso (`AutenticacaoTask`), e outra para realizar as consultas (`TwitterTask`). Durante a operação de consulta, vamos exibir um `ProgressDialog` na tela e, ao final do processamento, vamos atualizar a `ListView` com os *tweets* encontrados. A figura a seguir demonstra como ficará a nossa caixa de diálogo informando que a operação está em andamento.

Figura 6.2: Pesquisa no Twitter

Antes de realizar requisições de consulta, precisamos obter um token de acesso. Vamos fazer isso numa `AsyncTask` definida dessa maneira:

```
private class AutenticacaoTask
    extends AsyncTask<Void, Void, Void> {

}
```

Essa `AsyncTask` não receberá nenhum parâmetro e não retornará nenhum valor. Apenas atualizará o valor de uma variável da `activity` com o token de acesso. A requisição do token será realizada no método `doInBackground` e ela deve conter uma chave codificada em base 64, criada a partir da `key` e do `secret` da sua aplicação. Veja a seguir como fica o código para obter o token de acesso:

```
private class AutenticacaoTask
    extends AsyncTask<Void, Void, Void>{
```

```
    @Override
    protected Void doInBackground(Void... params) {
        try {
            Map<String, String> data =
                new HashMap<String, String>();
            data.put("grant_type", "client_credentials");
            String json = HttpRequest
                .post("https://api.twitter.com/oauth2/token")
                .authorization("Basic "+ gerarChave())
                .form(data)
                .body();

            JSONObject token = new JSONObject(json);
            accessToken = token.getString("access_token");
        } catch (Exception e) {
            return null;
        }
        return null;
    }

    private String gerarChave()
                    throws UnsupportedEncodingException{
        String key = "sua key";
        String secret = "seu secret";
        String token = key + ":" + secret;
        String base64 = Base64.encodeToString(token.getBytes(),
                                Base64.NO_WRAP);
        return base64;
    }
}
```

Nas linhas 6 e 7, configuramos os valores necessários para solicitar um token. Na linha 9, informamos a `url` do serviço do Twitter que desejamos. Na seguinte, informamos a chave gerada como forma de autenticação no Twitter para obtenção do token de acesso. Já na linha 14, criamos um novo objeto JSON, a partir da resposta enviada pelo Twitter.

O `JSONObject` mantém uma estrutura que representa o JSON e permite a navegação entre os elementos. Dessa forma, obtemos o token de acesso.

Na linha 15, atribuímos o token para uma variável da `activity` para utilizá-lo posteriormente. O método utilitário

`gerarChave` segue o estabelecido pelo Twitter, para criar uma chave para a requisição. Vamos disparar a `AutenticacaoTask` assim que atividade é iniciada para obter o token de acesso. Veja como:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    lista = (ListView) findViewById(R.id.lista);
    texto = (EditText) findViewById(R.id.texto);

    new AutenticacaoTask().execute();
}
```

Partindo para realizar a consulta, crie outra classe privada chamada `TwitterTask`. Começamos definindo que o `Params` dessa `AsyncTask` é do tipo `String`, pois iremos passar os termos da pesquisa que deve ser realizada. Já o `Progress` é definido como `Void`, ou seja, não faremos a atualização de progresso, e o tipo de retorno é um `String[]` com os *tweets* encontrados.

```
private class TwitterTask
    extends AsyncTask<String, Void, String[]> {

}
```

No método `onPreExecute` exibimos um `ProgressDialog` simples, e no `onPostExecute` fechamos este *dialog*, após construir um `ArrayAdapter` com os resultados e atribuí-lo a nossa `ListView`.

```
private class TwitterTask
    extends AsyncTask<String, Void, String[]> {

    ProgressDialog dialog;

    @Override
    protected void onPreExecute() {
        dialog = new ProgressDialog(TwitterSearchActivity.this);
        dialog.show();
    }
```

```
    @Override
    protected void onPostExecute(String[] result) {
        if(result != null){
            ArrayAdapter<String> adapter =
                new ArrayAdapter<String>(getBaseContext(),
                android.R.layout.simple_list_item_1, result);
            lista.setAdapter(adapter);
        }
        dialog.dismiss();
    }
}
```

Para implementar o método `doInBackground`, precisamos realizar uma requisição HTTP para uma URL, que será montada a partir dos termos de busca informados como parâmetro. Em seguida, processaremos a resposta obtida que está em formato JSON, criando um `String[]` com os dados dos *tweets*. Também para esse tipo de requisição, usaremos o framework `http-request`. Confira a implementação do método a seguir:

```
@Override
protected String[] doInBackground(String... params) {
    try {

        String filtro = params[0];

        if(TextUtils.isEmpty(filtro)){
            return null;
        }

        String urlTwitter =
            "https://api.twitter.com/1.1/search/tweets.json?q=";

        String url = Uri.parse(urlTwitter + filtro).toString();

        String conteudo = HttpRequest.get(url)
                .authorization("Bearer " + accessToken)
                .body();

        JSONObject jsonObject = new JSONObject(conteudo);

        JSONArray resultados =
            jsonObject.getJSONArray("statuses");
```

```
            //ainda falta processar os resultados

        } catch (Exception e) {
            Log.e(getPackageName(), e.getMessage(), e);
            throw new RuntimeException(e);
        }
    }
```

Por meio do `http-request`, fazemos uma nova requisição, agora para a URL do serviço de pesquisa do Twitter, incluindo o texto fornecido pelo usuário e também informando o token de acesso (linha 17). Em seguida, usamos o método `getJSONArray`, informando o atributo `"statuses"` que armazena os *tweets* em formato de `array`.

Para cada resultado obtido, temos que criar um novo `JSONObject` para recuperar os dados específicos que desejamos de cada *tweet*, que são o seu texto e o usuário que o realizou. Após os extrair, eles são armazenados em uma `String[]` que é retornada para ser colocada na `ListView`. Confira o código final:

```
@Override
protected String[] doInBackground(String... params) {
    try {

        String filtro = params[0];

        if(TextUtils.isEmpty(filtro)){
            return null;
        }

        String urlTwitter =
            "https://api.twitter.com/1.1/search/tweets.json?q=";

        String url = Uri.parse(urlTwitter + filtro).toString();

        String conteudo = HttpRequest.get(url)
            .authorization("Bearer " + accessToken)
            .body();

        JSONObject jsonObject = new JSONObject(conteudo);

        JSONArray resultados =
            jsonObject.getJSONArray("statuses");
```

```
        String[] tweets = new String[resultados.length()];

        for (int i = 0; i < resultados.length(); i++) {
            JSONObject tweet = resultados.getJSONObject(i);
            String texto = tweet.getString("text");
            String usuario = tweet.getJSONObject("user")
                                  .getString("screen_name");

            tweets[i] = usuario + " - " + texto;
        }

        return tweets;

    } catch (Exception e) {
        Log.e(getPackageName(), e.getMessage(), e);
        throw new RuntimeException(e);
    }
}
```

Para finalizar, agora precisamos iniciar a execução da `TwitterTask` quando o botão buscar for pressionado.

```
public void buscar(View v) {
    String filtro = texto.getText().toString();
    if(accessToken == null){
        Toast.makeText(this, "Token não disponível",
                            Toast.LENGTH_SHORT).show();
    }else{
        new TwitterTask().execute(filtro);
    }

}
```

Agora, é só executar a aplicação e testar! A figura a seguir mostra os resultados de uma pesquisa por `@android`. Para saber mais sobre a API de pesquisa do Twitter, visite https://dev.twitter.com/docs/using-search.

Figura 6.3: Pesquisa no Twitter

6.3 IMPLEMENTE UM SERVIÇO DE BACKGROUND

A primeira parte do aplicativo `TwitterSearch` está pronta. O objetivo desta seção é criar um serviço de *background* que faz pesquisas periódicas no Twitter em busca de menções ao seu usuário. Quando uma nova menção for encontrada, criaremos uma notificação na barra de *status* do dispositivo para alertar o usuário.

O `Service` é um componente da plataforma Android que pode executar tarefas de longa duração em plano de fundo e que não possui interface gráfica. Ele pode ser iniciado por qualquer outro tipo de componente (uma `Activity` ou outro `Service`, por exemplo) e pode continuar em execução mesmo que o componente que o iniciou seja destruído. Alguns exemplos de uso de `Service` incluem: fazer downloads, tocar uma música e acessar um serviço remoto.

Por padrão, o `Service` é executado no mesmo processo e na `thread` principal da aplicação. Portanto, se ele realiza operações bloqueantes, o desempenho da aplicação pode ficar comprometido, sendo necessário criar no `Service` uma nova `thread` para executar essas operações.

Um `Service` pode assumir duas formas, `started` e `bound`, que têm relação com a forma como ele é iniciado.

Um `Service` é `started` quando foi iniciado explicitamente por algum outro componente, por meio do método `Context.startService`. Neste caso, o `Service` é executado em plano de fundo indefinidamente, mesmo que o componente que o iniciou seja destruído. Esse tipo de `Service` é geralmente utilizado para realizar operações que não retornam resultados para quem o invocou. Quando a operação finalizar, o `Service` deve parar a si mesmo, usando o método `stopSelf`.

Ele é `bound` quando for iniciado por um componente, por meio do método `Context.bindService`. Essa forma de `Service` permite a interação entre componentes, como o envio de requisições e obtenção de respostas, em uma espécie de cliente-servidor. Deste modo, também é possível realizar a comunicação entre processos (IPC). Vários componentes podem fazer *bind* para o mesmo `Service`, que continuará existindo enquanto houver clientes (componentes) conectados.

O `Service` mais adequado para a nossa necessidade de fazer pesquisas no Twitter em plano de fundo é o `started`, pois, uma vez iniciado, ele será executado indefinidamente, além de não retornar dados para o componente que o invocou. A ideia é que a cada menção no Twitter, o `Service` crie notificações na barra de *status* para alertar o usuário. Para realizarmos isso, criamos uma nova classe chamada `NotificacaoService`, estendendo de `android.app.Service`:

```
public class NotificacaoService extends Service{
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

O método `onBind` deve ser implementado. Ele é invocado, quando algum componente deseja realizar o *bind* com este `Service` e deve retornar uma implementação de `IBinder` que será a interface utilizada para a comunicação entre os componentes. Como no nosso caso não permitiremos o *bind*, pois nosso `Service` é `started`, o método `onBind` retorna `null`. Precisamos sobrescrever o método `onStartCommand` que responde quando algum componente inicia o `Service`, por meio do método `Context.startService`.

```
@Override
public int onStartCommand(Intent intent, int flags,
        int startId) {
    //nossa implementação será feita aqui
    return START_STICKY;
}
```

O método `onStartCommand` retorna um `int` que indica como o serviço deve ser reiniciado pelo Android quando, por algum motivo, ele for encerrado. O Android pode encerrar (*kill*) um `Service`, caso o sistema esteja com falta de recursos, e recriá-lo posteriormente. A *flag* `START_STICKY` indica que o `Service` deve ser reiniciado e o método `onStartCommand` deve ser invocado novamente, mesmo que não haja nenhuma `Intent` pendente de processamento.

Existem outras *flags*, como a `START_NOT_STICKY`, que indicam que o `Service` só deve ser reiniciado se houver `Intents` pendentes, e a `START_REDELIVER_INTENT` para que o `Service` seja reiniciado com a última `Intent` enviada.

No método `onStartCommand`, devemos criar uma nova

`thread` para realizar a consulta por menções no Twitter e fazer com que ela seja executada periodicamente, de 10 em 10 minutos, por exemplo. Para fazer isso, usaremos a classe `ScheduledThreadPoolExecutor`, que permite que uma instância de `Runnable` seja executada de tempos em tempos, apenas configurando a sua execução, utilizando o método `scheduleAtFixedRate`.

Então, instanciaremos um novo `ScheduledThreadPoolExecutor`, passando no seu construtor a quantidade de `threads` que devem ser mantidas no *pool*. No nosso caso, apenas uma `thread` basta. Nas linhas seguintes, definimos qual será o atraso (*delay*) inicial para o início da execução; `0` indica que ela deve iniciar imediatamente.

Em seguida, definimos o período de tempo para que a operação seja realizada. Escolhemos que a `thread` deve ser executada de 10 em 10 minutos. Obviamente que, em cenários mais reais, provavelmente esse valor será configurado pelo usuário e lido das `SharedPreferences`. E, por fim, fazemos o agendamento da execução da `NotificacaoTask` que é uma classe privada que definiremos a seguir. Lembrando que você precisará obter previamente o token de acesso do Twitter para realizar as consultas. Você pode reaproveitar o código da `AutenticacaoTask` e fazer o mesmo aqui, iniciando a `AsyncTask` no `onStartCommand`.

```
@Override
public int onStartCommand(Intent intent, int flags,
        int startId) {
    ScheduledThreadPoolExecutor pool =
        new ScheduledThreadPoolExecutor(1);
    long delayInicial = 0;
    long periodo = 10;
    TimeUnit unit = TimeUnit.MINUTES;
    pool.scheduleAtFixedRate(new NotificacaoTask(),
                            delayInicial, periodo,unit);
    return START_STICKY;
}
```

Agora precisamos implementar a `NotificacaoTask`, que será responsável por realizar a tarefa. Ela implementará a interface `Runnable`, o que torna possível que ela seja executada pelo `ScheduledThreadPoolExecutor`.

A implementação do método `run` é bastante parecida com a que já realizamos para buscar os resultados no Twitter, a partir dos termos informados pelo usuário. Vamos, assim, às diferenças. Definimos uma variável com os termos que devem ser usados na primeira vez que a pesquisa for realizada. Inclua o seu usuário do Twitter lá. Depois, acrescentamos um método que verifica se há conectividade para dar prosseguimento à execução do método, chamado `estaConectado`, que implementaremos em seguida.

```
private class NotificacaoTask implements Runnable {
    private String baseUrl =
        "https://api.twitter.com/1.1/search/tweets.json";
    private String refreshUrl = "?q=@android";

    @Override
    public void run() {
        if (!estaConectado()) {
            return;
        }
        try {
            String conteudo =
                    HttpRequest.get(baseUrl + refreshUrl)
                    .authorization("Bearer " + accessToken)
                    .body();

            JSONObject jsonObject = new JSONObject(conteudo);
            refreshUrl = jsonObject.getString("refresh_url");

            JSONArray resultados =
                jsonObject.getJSONArray("statuses");

            for (int i = 0; i < resultados.length(); i++) {
                JSONObject tweet = resultados.getJSONObject(i);
                String texto = tweet.getString("text");
                String usuario = tweet.getJSONObject("user")
                                      .getString("screen_name");
                criarNotificacao(usuario, texto, i);
            }
```

```
        } catch (Exception e) {
            Log.e(getPackageName(), e.getMessage(), e);
        }
    }
}
```

Quando uma pesquisa é realizada, o Twitter retorna uma URL de atualização, contendo a pesquisa original e um parâmetro `since_id` que indica qual foi o último resultado retornado. Então, na `NotificacaoTask` atualizamos a variável `refreshUrl` para realizar as consultas posteriores, obtendo apenas as novas menções. Os resultados obtidos são percorridos e para cada um deles é criada uma nova notificação. Veremos como fazer isso na seção seguinte.

Agora, precisamos implementar o método `estaConectado`, que, por meio do `ConnectivityManager`, poderá obter informações variadas sobre o estado das conexões, sejam elas WI-FI, 3G etc. No nosso caso, recuperamos as informações da conexão ativa e, na linha seguinte, retornamos o valor que indica se o dispositivo está ou não conectado a uma rede de dados.

```
public class NotificacaoService extends Service {
    //demais códigos existentes
    private boolean estaConectado() {
        ConnectivityManager manager =
            (ConnectivityManager) getSystemService(
                Context.CONNECTIVITY_SERVICE);

        NetworkInfo info = manager.getActiveNetworkInfo();

        return info.isConnected();
    }
}
```

Agora, para ficar completo, precisamos implementar o método `criarNotificacao`.

6.4 CRIE NOTIFICAÇÕES NA BARRA DE STATUS

A barra de status do Android é o local onde as notificações são exibidas. Elas podem ser do próprio sistema, como por exemplo, um aviso sobre o descarregamento da bateria, ou podem ter sido criadas por qualquer aplicativo.

A primeira figura adiante mostra como deve ficar as notificações do nosso aplicativo `TwitterSearch` na barra de *status*. Quando o usuário selecionar alguma das notificações, devemos mostrar o *tweet* em uma nova *activity* da nossa aplicação. A segunda figura mostra uma visualização simples do *tweet*.

Figura 6.4: Notificações na barra de status

Figura 6.5: Notificações na barra de status

Para construir as notificações, implementamos na classe `NotificacaoService` o método `criarNotificacao`. Este recebe como parâmetro o texto do *tweet* e o usuário que o realizou, além de um identificador para a notificação.

O método vai receber o usuário, o texto e o `id` do tweet:

```
private void criarNotificacao(String usuario,
        String texto, int id) { }
```

Definimos qual será o ícone da notificação, recuperamos do `strings.xml` o texto que aparecerá na barra de *status*, obtemos um `long` que representa a data da notificação e também construímos o título da notificação, que será algo como "alguém mencionou você". Em seguida, criamos uma `Intent` que será utilizada para iniciar uma *activity* que exibirá os dados do *tweet*,

incluídos como extras da `Intent`.

Usaremos a `PendingIntent` para iniciar a `TweetActivity`, quando o usuário selecionar uma notificação. Basicamente, a `PendingIntent` permite que outro componente execute a ação definida anteriormente pela `Intent`, como se fosse a própria aplicação que a criou. Utilizaremos esse recurso para vincular cada notificação a uma `Intent` que exibe os dados do *tweet*.

```
private void criarNotificacao(String usuario,
            String texto, int id) {
    int icone = R.drawable.ic_launcher;
    String aviso = getString(R.string.aviso);
    long data = System.currentTimeMillis();
    String titulo = usuario + " " + getString(R.string.titulo);

    Context context = getApplicationContext();
    Intent intent = new Intent(context, TweetActivity.class);
    intent.putExtra(TweetActivity.USUARIO, usuario.toString());
    intent.putExtra(TweetActivity.TEXTO, texto.toString());

    PendingIntent pendingIntent =
            PendingIntent.getActivity(context, id, intent,
                            Intent.FLAG_ACTIVITY_NEW_TASK);

}
```

Criamos uma `PendingIntent`, recebendo como parâmetro o contexto, um identificador para a requisição (atualmente não utilizado pelo Android), a `Intent` desejada e uma *flag* de inicialização.

Agora, vamos criar uma nova notificação com o ícone, o texto de aviso e a data previamente definidos. Usaremos a *flag* `FLAG_AUTO_CANCEL` para indicar que, ao ser selecionada essa notificação, vamos sair da lista de notificações. Adicionalmente, configuramos para que ela provoque a vibração do dispositivo, toque o som padrão ou, ainda, ative a iluminação de aviso.

Vamos usar o método `setLatestEventInfo` para incluir as

informações da notificação que são o usuário e texto do *tweet* e a `PendingIntent` que será executada. Agora que a notificação está pronta, utilizamos o `NotificationManager` para efetivamente a colocar na barra de *status*.

```
private void criarNotificacao(String usuario,
            String texto, int id) {
    // Recupera as informações e cria a PendingIntent

    Notification notification =
        new Notification(icone, aviso, data);
    notification.flags = Notification.FLAG_AUTO_CANCEL;
    notification.flags = Notification.FLAG_AUTO_CANCEL;
    notification.defaults |= Notification.DEFAULT_VIBRATE;
    notification.defaults |= Notification.DEFAULT_LIGHTS;
    notification.defaults |= Notification.DEFAULT_SOUND;
    notification.setLatestEventInfo(context, titulo,
                                texto, pendingIntent);

    String ns = Context.NOTIFICATION_SERVICE;
    NotificationManager notificationManager =
            (NotificationManager) getSystemService(ns);
    notificationManager.notify(id, notification);
}
```

Precisamos colocar também as mensagens no `strings.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">TwitterSearch</string>
    <string name="buscar">Buscar</string>
    <string name="aviso">Você foi mencionado!</string>
    <string name="titulo">mencionou você</string>
</resources>
```

A *activity* criada para a exibição dos *tweets* é bastante simples e não possui nada de diferente do que já foi apresentado no decorrer no livro. Confira o arquivo de layout `tweet.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

```
    <TextView
        android:id="@+id/usuario"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/texto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

E a classe `TweetActivity`:

```
public class TweetActivity extends Activity {
    public static final String TEXTO = "texto";
    public static final String USUARIO = "usuario";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tweet);

        TextView usuarioTextView =
                (TextView) findViewById(R.id.usuario);
        TextView textoTextView =
                (TextView) findViewById(R.id.texto);

        String usuario = getIntent().getStringExtra(USUARIO);
        String texto = getIntent().getStringExtra(TEXTO);

        usuarioTextView.setText(usuario);
        textoTextView.setText(texto);
    }
}
```

Para finalizar a implementação do `Service` e das notificações, é necessário declarar o `Service` no `AndroidManifest.xml`, assim como a `TweetActivity`. Também é preciso inserir uma permissão referente ao acesso às informações de rede e outra para ativar a vibração do dispositivo. Veja como ficou:

```
<!-- permissões necessárias -->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
```

```
    android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.VIBRATE"/>

<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <!-- declarações existentes -->
    <service android:name=".NotificacaoService" />
    <activity android:name=".TweetActivity" />
</application>
```

Ainda não é possível ver o serviço implementado executando, pois, em nenhum momento, fizemos a sua inicialização chamando o método `Context.startService`. A ideia é que isso não seja feito pelo usuário, mas sim pelo próprio Android, quando o dispositivo for iniciado.

6.5 UTILIZE UM BROADCASTRECEIVER PARA INICIAR O SERVICE

A última parte da nossa aplicação que usa o Twitter consiste em implementar um outro tipo de componente do Android, chamado de *Broadcast Receiver*. Esse componente pode responder a eventos propagados (*broadcasts*) pelo sistema e também por outros aplicativos. Assim como o componente `Service`, o `BroadcastReceiver` também não possui interface gráfica e geralmente interage com o usuário por meio de notificações.

Faremos uma implementação que representa bem o papel do `BroadcastReceiver`, que é receber um evento e processá-lo, geralmente delegando a execução para outro componente.

Crie uma nova classe com nome de `StartupReceiver`, estendendo de `BroadcastReceiver`. Temos que implementar apenas o método `onReceive` que recebe como parâmetro o contexto e o evento, que, na verdade, é uma `Intent`, algo que já conhecemos bem. Nesse método, iniciaremos o

`NotificacaoService` . O código a seguir demostra a implementação do `StartupReceiver` :

```
public class StartupReceiver extends BroadcastReceiver{
    @Override
    public void onReceive(Context context, Intent intent) {
        Intent it =
            new Intent(context, NotificacaoService.class);
        context.startService(it);
    }
}
```

No nosso caso, o evento que queremos capturar é o `android.intent.action.BOOT_COMPLETED` , que indica que o dispositivo foi inicializado. Como os eventos, na verdade, são `Intents` , fica fácil imaginar que precisaremos incluir um `intent filter` para capturá-los. No `AndroidManifest.xml` , faremos a declaração do `BroadcastReceiver` e do filtro necessário, além, é claro, de uma permissão para receber o evento, dessa forma:

```
<!-- permissões já existentes -->
<uses-permission
    android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <!-- declarações existentes -->
    <receiver android:name=".StartupReceiver" >
        <intent-filter>
            <action
    android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>
</application>
```

Agora a aplicação `TwitterSearch` está pronta! Execute-a normalmente.

Na primeira execução, o Android Studio instalará o aplicativo; no entanto, o evento de *boot* já foi disparado; logo, o aplicativo não irá iniciar o `Service` . Feche o emulador e o inicie novamente, mas,

desta vez, pelo AVD Manager (menu `Tools > Android > AVD Manager`). Agora o serviço deve ser iniciado, e, para testar, basta tuitar algo com os termos que você especificou.

> **VERIFICANDO SERVIÇOS EM EXECUÇÃO**
>
> No Android, é possível verificar quais aplicativos e serviços estão sendo executados através do menu `Configurações` , opção `Aplicativos` , aba `Em execução` .

6.6 CONCLUSÃO

Acabamos de aprender um pouco dos serviços REST que utilizam o formato JSON para transmitir dados. Experimentamos a Google Books API, que disponibiliza informações sobre livros, por meio de um serviço REST, e implementamos uma aplicação de exemplo que realiza buscas no Twitter e faz notificações quando alguém nos menciona em algum *tweet*.

Para as implementações, usamos `AsyncTask` e os componentes `Service` e `BroadcastReceiver` da plataforma Android, além de agendar a execução de `thread` com o `ScheduledThreadPoolExecutor` . Todos esses itens são frequentemente utilizados quando trabalhamos com serviços remotos.

Você pode aprender muito mais sobre REST através de livros específicos, como o *RESTful Web services*, do Leonard Richardson e Sam Ruby, ou então, do bom material disponibilizado pela IBM, em seu blog de desenvolvedores: http://www.ibm.com/developerworks/webservices/library/ws-restful.

CAPÍTULO 7

UTILIZE GOOGLE APIS E CRIE FUNCIONALIDADES INTERESSANTES

No capítulo *Integração de aplicações Android com serviços REST*, fizemos a integração com o Twitter e criamos uma aplicação que notificava o usuário a cada novo *tweet* que surgisse sobre um determinado assunto. Mas, e se quisermos colocar em nossa aplicação um serviço de calendário, para podermos guardar em nossas agendas as viagens que teremos que fazer? Como poderíamos implementar essa funcionalidade?

O Google disponibiliza diversos serviços relacionados aos seus produtos que podem ser utilizados gratuitamente por nós, desenvolvedores. Esses serviços também são conhecidos como Google APIs. Alguns exemplos são a Places API, que permite obter informações sobre determinada localidade; a Tasks API, que permite interações com o Google Tasks; a Google Maps API; e a Drive API, para integração com o Google Drive.

De forma geral, estes serviços podem ser acessados diretamente via HTTP, sendo responsabilidade do aplicativo do cliente realizar os procedimentos de autenticação, executar a requisição e processar a resposta do serviço. No entanto, o Google disponibiliza uma série de bibliotecas, em diversas linguagens de programação, conhecidas como *Google APIs Client Libraries*, que facilitam o uso desses

serviços.

Neste capítulo, veremos como utilizar uma conta do Google para autenticar um usuário e incluiremos essa funcionalidade no login do nosso aplicativo BoaViagem. Além disso, faremos uma integração do aplicativo com a agenda do usuário no Google Calendar. Assim que ele criar uma nova viagem, registraremos em sua agenda um novo evento compreendendo essa data, usando a Calendar API. Para realizar essas implementações são necessários alguns procedimentos de preparação, que veremos nas seções a seguir.

> **Não conhece o Google Calendar?**
>
> O Google Calendar, ou Google Agenda como é chamado no Brasil, é um serviço gratuito com interface web, no qual é possível registrar e controlar eventos e compromissos, além de poder compartilhá-los com outras pessoas. Para utilizar o serviço, é necessário ter uma conta no Google. Conheça-o em http://www.google.com/calendar.

7.1 CONFIGURE O ADD-ON GOOGLE APIS

O *add-on* Google APIs é uma extensão do Android SDK, que disponibiliza essencialmente uma API externa para trabalhar com mapas, além de outros componentes e serviços do Google que rodam no Android. Em nosso caso, estamos interessados no serviço de contas de usuário, que usaremos para autenticação. É importante ressaltar que esse add-on nada tem a ver com os diversos serviços disponibilizados pelo Google, tais como a Google+ API, Calendar API e CustomSearch API, e as suas respectivas bibliotecas de acesso.

Para obter o add-on, acesse o Android SDK Manager e selecione o add-on Google APIs da versão do Android que você está utilizando, e também a imagem do emulador correspondente que contém a Google APIs, conforme demonstra a figura a seguir. Geralmente, ao instalar o Android Studio, ele já trará esses dois itens; portanto, esse passo só será necessário quando você precisar baixar uma nova versão do Android.

Figura 7.1: Download do pacote Google APIs

Para usar o serviço do Google Calendar, é necessário que o usuário possua uma conta do Google. Esta também precisa estar registrada no dispositivo para que possamos utilizá-la tanto para autenticá-lo quanto para solicitar acesso à sua agenda.

Inicie o AVD criado com o add-on Google APIs. Com o emulador já iniciado, acesse o menu do dispositivo, selecione `Configurações` e, em seguida, a opção `Contas` e

`Sincronização`. Escolha `Adicionar nova conta` e siga as instruções. Você pode usar a sua própria conta do Google ou criar uma especificamente para testes.

7.2 ADICIONE BIBLIOTECAS AUXILIARES

O Google, além de disponibilizar APIs para acessar remotamente os seus serviços, também concede um conjunto de bibliotecas para acessá-los em diversas linguagens de programação diferentes. Utilizaremos algumas delas para nos ajudar na autenticação do usuário e na integração com o Google Calendar. Acesse o site https://developers.google.com/api-client-library/java/apis/calendar/v3 e faça o download da última versão da biblioteca Calendar API.

Ao fazer o download e descompactar o arquivo, além da própria Calendar API, também teremos as suas dependências, localizadas na pasta `libs`. Como elas variam conforme a versão, consulte o arquivo `readme.html` para checar quais dependências devem ser adicionadas ao seu projeto Android.

Quando trabalhamos com bibliotecas de terceiros, temos basicamente duas opções para adicioná-las ao nosso projeto: colocando na pasta `libs`, ou declarando uma dependência do Gradle quando esta já existe e está disponível em algum repositório. No Android Studio, se ainda não existir, crie uma nova pasta no diretório `app` com o nome de `libs` e copie para lá os arquivos listados como dependência da Calendar API.

Utilize a visão `Project` do Android Studio para facilitar a operação. A estrutura do projeto ficará conforme demonstra a figura:

Figura 7.2: Pasta libs do projeto

Agora é necessário indicar para o Gradle — o responsável por resolver as dependências do projetos — que ele deve considerar todos os arquivos `.jar` existentes na pasta `libs`. Para isso, basta acrescentar a linha a seguir no arquivo `build.gradle` do módulo `app`, na seção `dependencies`.

```
compile fileTree(dir: 'libs', include: ['*.jar'])
```

7.3 ADICIONE AS PERMISSÕES NECESSÁRIAS

É necessário adicionar algumas permissões no `AndroidManifest.xml` para que possamos acessar a Internet e recuperar as contas armazenadas no dispositivo. As permissões necessárias são as seguintes:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
    android:name="android.permission.USE_CREDENTIALS" />
<uses-permission
    android:name="android.permission.GET_ACCOUNTS" />
<uses-permission
    android:name="android.permission.MANAGE_ACCOUNTS" />
```

7.4 REGISTRE A APLICAÇÃO NO GOOGLE

Além das bibliotecas auxiliares para realizar a integração com o Google Calendar, precisaremos de mais duas coisas. A primeira é obter uma chave (API *key*) para identificar o nosso aplicativo junto ao Google. Já a segunda é criar um *client ID* que será usado para que o usuário autorize o acesso aos dados da sua agenda. Essa autorização será realizada por meio do protocolo OAuth 2.0; um protocolo para autorização e autenticação, utilizado por diversos serviços conhecidos, como Twitter e Facebook.

O registro da aplicação para a obtenção das chaves é feito através do Google Console. É necessário possuir uma conta do Google para realizar esse procedimento. Então, faremos isso agora.

Acesse o site https://code.google.com/apis/console/ e clique em `Create Project` .

Figura 7.3: Google Console

Na tela seguinte, ative a Calendar API clicando no botão `off`

da coluna `Status` .

Figura 7.4: Ativando o serviço

No menu localizado do lado esquerdo, selecione a opção `API Access` . Na parte inferior da página apresentada está listada a API *key* que utilizaremos para identificar nosso aplicativo no Google. A figura a seguir destaca a API *key*.

Figura 7.5: API key

Nessa mesma página, também teremos uma opção para criar uma "OAuth 2.0 client ID". Ao selecioná-la, será necessário informar o nome do aplicativo que será apresentado ao usuário, quando for solicitada autorização para acessar seus dados da agenda. Opcionalmente, podemos informar a URL do logotipo do produto.

Ao selecionar a opção `Next`, devemos informar o tipo da aplicação. No nosso caso, a opção adequada é a `Installed Application`, pois a nossa aplicação será instalada em um dispositivo. As imagens a seguir demonstram esses passos.

Figura 7.6: Criando client ID — Passo 1

Figura 7.7: Criando client ID — Passo 2

E pronto, você já registrou sua aplicação no Google!

7.5 AUTENTIQUE O USUÁRIO COM A CONTA DO GOOGLE

Com o ambiente preparado, agora podemos iniciar a implementação das novas funcionalidades! Iniciaremos realizando a autenticação do usuário com base na sua conta do Google, registrada no dispositivo. Esta abordagem é interessante, pois evita que nosso aplicativo tenha que manter uma base própria de usuários, além de trazer comodidade, uma vez que eles podem usar os seus logins e senhas já existentes.

Na classe `BoaViagemActivity` , vamos substituir a autenticação existente e incluir essa nova implementação. A ideia é que o usuário forneça o login e senha de sua conta Google. A partir dessas informações, utilizaremos o `AccountManager` do Android e solicitaremos para que as credenciais sejam confirmadas.

O `AccountManager` suporta vários tipos de conta, como as do Facebook e da Microsoft Exchange. No momento, nos interessa apenas as do tipo Google, por isso usaremos a classe `GoogleAccountManager` que já traz facilidades para lidarmos com ela.

No método `onCreate` da `BoaViagemActivity` , vamos instanciar o `GoogleAccountManager` , passando o contexto como parâmetro. Também realizaremos algumas alterações no método `onCreate` , para utilizar um arquivo de preferências global para o aplicativo, e moveremos a constante `MANTER_CONECTADO` para a classe de constantes.

Incluímos um novo método `iniciarDashboard` que será usado também em outras partes do código:

```
import static br.com.casadocodigo.boaviagem.Constantes.*

// novos atributos
```

```
private SharedPreferences preferencias;
private GoogleAccountManager accountManager;
private Account conta;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.login);

    accountManager = new GoogleAccountManager(this);

    usuario = (EditText) findViewById(R.id.usuario);
    senha = (EditText) findViewById(R.id.senha);
    manterConectado =
        (CheckBox) findViewById(R.id.manterConectado);

    preferencias =
        getSharedPreferences(PREFERENCIAS, MODE_PRIVATE);
    boolean conectado = preferencias
        .getBoolean(MANTER_CONECTADO, false);

    if(conectado){
        iniciarDashboard();
    }
}

private void iniciarDashboard(){
    startActivity(new Intent(this, DashboardActivity.class));
}

public class Constantes {
    // novas constantes
    public static final String PREFERENCIAS =
        "preferencias_globais";
    public static final String MANTER_CONECTADO =
        "manter_conectado";
}
```

Vamos alterar a implementação do método `entrarOnClick`, que agora passará a invocar o método `autenticar`.

```
public void entrarOnClick(View v){
    String usuarioInformado = usuario.getText().toString();
    String senhaInformada = senha.getText().toString();

    autenticar(usuarioInformado, senhaInformada);
}
```

No método autenticar , caso não seja possível localizar a conta, emitimos uma mensagem de aviso para o usuário. Para realizar a autenticação, construímos um Bundle , informando as credenciais do usuário, e solicitamos sua verificação, invocando o método confirmCredentials do AccountManager .

```
private void autenticar(final String nomeConta, String senha) {

    conta = accountManager.getAccountByName(nomeConta);

    if(conta == null){
        Toast.makeText(this, R.string.conta_inexistente,
                                Toast.LENGTH_LONG).show();
        return;
    }

    Bundle bundle = new Bundle();
    bundle.putString(AccountManager.KEY_ACCOUNT_NAME, nomeConta);
    bundle.putString(AccountManager.KEY_PASSWORD, senha);

    accountManager.getAccountManager()
        .confirmCredentials(conta , bundle, this,
            new AutenticacaoCallback(), null);
}
```

Como parâmetro para o método confirmCredentials , precisamos passar a conta, o bundle com as credenciais, o contexto e um callback que será invocado quando a operação for realizada, além de um handler (nulo, em nosso exemplo).

O método confirmCredentials não retorna nada de imediato; ele será executado de forma assíncrona e, quando a operação solicitada finalizar, o callback será invocado. É lá que verificaremos se a autenticação ocorreu com sucesso.

Optamos por implementar esse callback como uma classe privada, e temos que implementar a interface AccountManagerCallback :

```
private class AutenticacaoCallback
        implements AccountManagerCallback<Bundle> {
```

```
    @Override
    public void run(AccountManagerFuture<Bundle> future) { }
}
```

Em seguida, obtemos o `bundle` que contém os resultados da operação de confirmação de credenciais.

```
private class AutenticacaoCallback
        implements AccountManagerCallback<Bundle> {

    @Override
    public void run(AccountManagerFuture<Bundle> future) {
        try {

            Bundle bundle = future.getResult();


        } catch (OperationCanceledException e) {
            // usuário cancelou a operação
        } catch (AuthenticatorException e) {
            // possível falha no autenticador
        } catch (IOException e) {
            // possível falha de comunicação
        }
    }
}
```

Em seguida, verificamos se a operação foi realizada com sucesso; ou seja, se o login e senha foram informados corretamente, avaliando se o conteúdo armazenado na chave `KEY_BOOLEAN_RESULT` é verdadeiro. Em caso de falha, alertamos o usuário; caso contrário, a dashboard do aplicativo é iniciada.

```
private class AutenticacaoCallback
        implements AccountManagerCallback<Bundle> {

    @Override
    public void run(AccountManagerFuture<Bundle> future) {
        try {

            Bundle bundle = future.getResult();
            if(bundle.getBoolean(
                AccountManager.KEY_BOOLEAN_RESULT)) {

                iniciarDashboard();
```

```
            } else {
                Toast.makeText(getBaseContext(),
                        getString(R.string.erro_autenticacao),
                        Toast.LENGTH_LONG).show();
            }

        } catch (OperationCanceledException e) {
            //usuário cancelou a operação
        } catch (AuthenticatorException e) {
            //possível falha no autenticador
        } catch (IOException e) {
            //possível falha de comunicação
        }
    }
}
```

Pronto! Execute a aplicação e autentique-se com a mesma conta adicionada ao dispositivo!

7.6 SOLICITE AUTORIZAÇÃO PARA O GOOGLE CALENDAR

Para realizar a integração, vamos precisar de uma autorização concedida pelo usuário para que o nosso aplicativo possa acessar seus dados da agenda.

Essa autorização será feita utilizando o protocolo OAuth 2.0, que tem sido amplamente usado por serviços na web. Ele estabelece uma forma simples e segura de autorizar o acesso a um dado recurso por um determinado aplicativo, não sendo necessário que ele tenha acesso às credenciais do usuário.

Basicamente, os passos realizados para uma autorização via OAuth 2.0 são os seguintes:

1. O aplicativo requer ao serviço desejado autorização para acessar determinado recurso;
2. O serviço solicita o consentimento do usuário;
3. Se autorizado, o serviço fornece um *token* de acesso para o

aplicativo;
4. O aplicativo utiliza esse *token* para realizar as requisições.

O *client ID* que criamos anteriormente será usado nos passos 1 e 2 para identificar o nosso aplicativo. Como estamos utilizando as bibliotecas auxiliares disponibilizadas pelo Google, não precisaremos implementar todo esse procedimento. No aplicativo BoaViagem, aproveitaremos o momento do login para solicitar a autorização necessária, para acessar a agenda do usuário.

Quando fazemos o pedido de acesso, precisamos informar qual é o recurso desejado. No âmbito do OAuth, o que desejamos acessar é conhecido como "escopo", que no nosso caso é o Google Calendar, identificado como `oauth2:https://www.googleapis.com/auth/calendar` .

Precisaremos informar esse escopo quando fizermos a solicitação de um *token* de acesso. Vamos incluir este valor na classe `Constantes` . Aproveite também para incluir mais duas constantes que se referem à sua API *key* e ao nome da aplicação utilizado no registro do Google Console. Veja a seguir a definição das novas constantes:

```
public class Constantes {
    // novas constantes
    public static final String TOKEN_ACESSO = "token_acesso";
    public static final String NOME_CONTA = "nome_conta";
    public static final String APP_NAME = "BoaViagem";
    public static final String AUTH_TOKEN_TYPE =
            "oauth2:https://www.googleapis.com/auth/calendar";
    public static final String API_KEY =
            "sua_api_key_obtida_no_google_console";

    // constantes existentes
}
```

Assim como fizemos na autenticação, criaremos um novo método e um `callback` para tratar da autorização. Novamente usaremos o `AccountManager` , invocando o seu método

`getAuthToken` para recuperar um *token* de acesso. Como parâmetro, informaremos a conta e o escopo, além da *activity* e do `callback` que será invocado quando a operação terminar. Opcionalmente, podemos informar também um `bundle` e um `handler`. O método `solicitarAutorizacao` terá o seguinte código:

```
private void solicitarAutorizacao() {

    accountManager.getAccountManager()
            .getAuthToken(conta,
            Constantes.AUTH_TOKEN_TYPE,
            null,
            this,
            new AutorizacaoCallback(),
            null);
}
```

O `AutorizacaoCallback` será utilizado para tratar a resposta do pedido de autorização. O que precisaremos fazer é recuperar o *token* de acesso e o nome da conta, e gravar essas informações nas preferências do aplicativo para usarmos posteriormente.

```
private class AutorizacaoCallback
        implements AccountManagerCallback<Bundle> {

    @Override
    public void run(AccountManagerFuture<Bundle> future) {

        try {
            Bundle bundle = future.getResult();
            String nomeConta =
        bundle.getString(AccountManager.KEY_ACCOUNT_NAME);
            String tokenAcesso =
                bundle.getString(AccountManager.KEY_AUTHTOKEN);

        } catch (OperationCanceledException e) {
            // usuário cancelou a operação
        } catch (AuthenticatorException e) {
            // possível problema no autenticador
        } catch (IOException e) {
            // possível problema de comunicação
        }
    }
```

```
}
```

Em seguida, gravamos essas informações nas preferências e iniciamos a dashboard do aplicativo.

```
private class AutorizacaoCallback
    implements AccountManagerCallback<Bundle> {

    @Override
    public void run(AccountManagerFuture<Bundle> future) {

        try {
            Bundle bundle = future.getResult();
            String nomeConta =
        bundle.getString(AccountManager.KEY_ACCOUNT_NAME);
            String tokenAcesso =
                bundle.getString(AccountManager.KEY_AUTHTOKEN);

            gravarTokenAcesso(nomeConta, tokenAcesso);

            iniciarDashboard();

        } catch (OperationCanceledException e) {
            // usuário cancelou a operação
        } catch (AuthenticatorException e) {
            // possível falha no autenticador
        } catch (IOException e) {
            // possível falha de comunicação
        }
    }
}

private void gravarTokenAcesso(String nomeConta,
        String tokenAcesso) {
    Editor editor = preferencias.edit();
    editor.putString(NOME_CONTA, nomeConta);
    editor.putString(TOKEN_ACESSO, tokenAcesso);
    editor.commit();
}
```

No `AutenticacaoCallback`, quando a autenticação é realizada com sucesso, em vez de iniciar a dashboard, invocamos o método `solicitarAutorizacao`.

```
private class AutenticacaoCallback
        implements AccountManagerCallback<Bundle> {
    @Override
```

```java
    public void run(AccountManagerFuture<Bundle> future) {

        try {

            Bundle bundle = future.getResult();
            if(bundle.getBoolean(
                AccountManager.KEY_BOOLEAN_RESULT)) {

                solicitarAutorizacao();

            } else {
                Toast.makeText(getBaseContext(),
                        getString(R.string.erro_autenticacao),
                        Toast.LENGTH_LONG).show();
            }

        } catch (OperationCanceledException e) {
            // usuário cancelou a operação
        } catch (AuthenticatorException e) {
            // possível problema no autenticador
        } catch (IOException e) {
            // possível problema de comunicação
        }
    }
}
```

Com essas implementações, finalizamos a parte de autorização para utilizar o Google Calendar! Agora, quando você se autenticar no aplicativo, será perguntado se autoriza o acesso aos seus dados da agenda, como mostra a figura seguinte. Experimente!

Figura 7.8: Solicitação de autorização

7.7 TRATE A EXPIRAÇÃO DO TOKEN DE ACESSO

O *token* fornecido para acessar os serviços do Google geralmente expiram uma hora após a sua criação. No entanto, outros serviços, como Twitter e Facebook, que também possuem autorização via OAuth, podem estipular um período de tempo diferente para a validade do *token* de acesso. Nossas aplicações precisam tratar de sua expiração; caso contrário, teremos problemas de acesso não autorizado.

Existem diversas estratégias para lidar com essa expiração que variam de acordo com o cenário de uso e o tipo de aplicação (web, mobile etc.) que utiliza o *token*.

Uma alternativa é monitorar o tempo de expiração e revalidá-lo, antes que o tempo se esgote. Existem dois métodos na classe `Credential` que podem nos ajudar nisso: o `getExpirationTimeMilliseconds`, que retorna a validade do *token* em milissegundos; e o `getExpiresInSeconds()`, que devolve a quantidade de segundos que restam antes que ele expire.

Na aplicação BoaViagem, sempre obteremos um novo *token*, quando o usuário acessar a aplicação. Para fazer isso, será necessário invalidar o obtido anteriormente do `AccountManager`. O *token* já conhecido será retornado, e pode já estar expirado.

É importante frisar que obter um novo *token* não quer dizer que o usuário terá que autorizar o aplicativo novamente. O aplicativo continua autorizado e o usuário não notará que isso aconteceu.

A invalidação do *token* e a recuperação de um novo serão realizadas no método `solicitarAutorizacao`. Para invalidá-lo, invocamos o método `invalidateAuthToken` do `AccountManager`. O código que solicita um novo *token* permanece o mesmo já apresentado.

Ainda que o usuário tenha escolhido se manter conectado — ou seja, entrar direto sem informar as credenciais —, teremos que obter um *token* válido novamente. Portanto, alteramos o método `onCreate` para invocar o método `solicitarAutorizacao`, em vez de `iniciarDashboard`, quando a opção "Manter conectado" foi marcada. Com isso, teremos o seguinte código:

```
private void solicitarAutorizacao() {

    String tokenAcesso =
        preferencias.getString(TOKEN_ACESSO, null);
    String nomeConta =
        preferencias.getString(NOME_CONTA, null);

    if(tokenAcesso != null){
        accountManager.invalidateAuthToken(tokenAcesso);
```

```
        conta = accountManager.getAccountByName(nomeConta);
    }

    accountManager.getAccountManager()
            .getAuthToken(conta,
            Constantes.AUTH_TOKEN_TYPE,
            null,
            this,
            new AutorizacaoCallback(),
            null);
}

@Override
public void onCreate(Bundle savedInstanceState) {
  //códigos existentes
  if(conectado){
    solicitarAutorizacao();
   }
}
```

Pronto! Agora a nossa aplicação está utilizando uma conta do Google para autenticar o usuário e mantê-lo conectado.

7.8 CONHEÇA A CALENDAR API

A Calendar API permite desenvolver aplicações que interagem com a agenda do usuário, sendo possível criar novos eventos, alterar e excluir os que já existem, além de permitir a realização de buscas. Também é possível criar e remover calendários. A Calendar API possui alguns conceitos básicos:

- `Event` — é um evento de uma agenda (calendário) que possui informações, como: título, participantes e data de início e de fim;

- `Calendar` — representa uma única agenda do usuário. No Google Calendar, é possível criar várias agendas; uma é considerada como a principal e é identificada pelo nome da conta do Google, e as demais são classificadas como secundárias;

- `Calendar List` — é a lista de calendários do usuário, contendo o principal e os secundários;
- `Setting` — representa as preferências do usuário, como o seu fuso horário;
- `ACL` — é uma regra de acesso (compartilhamento) do calendário;
- `Color` — são as cores utilizadas para destacar os eventos e as agendas;
- `Free/busy` — conjunto de períodos nos quais os calendários não possuem nenhum evento.

Para cada conceito existe um recurso associado (`Event` , `Calendar` e outros). Os recursos são agrupados e disponibilizados pelo serviço através de coleções (`Events` , `Calendars` , `CalerdarList` , `Settings` e `ACL`). Este modelo se reflete na biblioteca cliente que dispõe de classes para representar esses recursos e coleções.

A documentação completa da Calendar API está disponível em http://code.google.com/apis/calendar/v3/using.html. Veremos a seguir algumas formas de utilizá-la.

7.9 ADICIONE EVENTOS NO GOOGLE CALENDAR

A autenticação e autorização estão prontas e não corremos mais o risco de usar um *token* de acesso inválido. Além disso, já vimos um pouco do que é a Calendar API.

Partiremos agora para a inclusão de um evento na agenda do usuário. Basicamente, o que precisaremos fazer é utilizar as classes

disponibilizadas pela biblioteca cliente da Calendar API, para criar os objetos necessários e invocar o método que realizará a inclusão dos dados na agenda. Toda a parte de comunicação HTTP, serialização dos dados e tratamento das respostas será feita pela biblioteca cliente.

Para continuar mantendo nosso código organizado, vamos criar uma nova classe para tomar conta dessa integração com o Google Calendar. Crie a classe `CalendarService` no pacote `br.com.casadocodigo.boaviagem.calendar` . De antemão, precisaremos do nome da conta e do *token* de acesso que serão utilizados, vamos recebê-los no construtor da classe:

```
public class CalendarService {

    private Calendar calendar;
    private String nomeConta;

    public CalendarService(String nomeConta,
            String tokenAcesso) {
        this.nomeConta = nomeConta;
        GoogleCredential credencial = new GoogleCredential();
        credencial.setAccessToken(tokenAcesso);
    }
}
```

O nome da conta será usado posteriormente para identificar qual calendário será utilizado, lembrando que este será o principal do usuário. Criamos também uma nova credencial do aplicativo e atribuímos a ela o *token* de acesso. Essa credencial será usada durante as requisições para o serviço.

A classe `Calendar` do pacote `com.google.api.services.calendar` representa o serviço propriamente dito e, através dela, conseguiremos manipular os eventos. Sua instanciação é feita por meio de um `builder` que depende de dois outros objetos pertencentes à Google API Client Library: o `HttpTransport` , para a comunicação HTTP, e o

`JsonFactory` , para serialização dos objetos. Além disso, devemos atribuir à nossa API *key* a credencial e o nome do aplicativo. Tudo isso realizado ainda no construtor:

```
public CalendarService(String nomeConta, String tokenAcesso) {
    this.nomeConta = nomeConta;
    GoogleCredential credencial = new GoogleCredential();
    credencial.setAccessToken(tokenAcesso);

    HttpTransport transport =
        AndroidHttp.newCompatibleTransport();
    JsonFactory jsonFactory = new GsonFactory();

    calendar = Calendar.builder(transport, jsonFactory)
            .setApplicationName(Constantes.APP_NAME)
            .setHttpRequestInitializer(credencial)
            .setJsonHttpRequestInitializer(
                new GoogleKeyInitializer(Constantes.API_KEY))
            .build();
}
```

A ideia é que um evento seja criado a partir das informações de uma viagem. Logo, criaremos um método no `CalendarService` que recebe uma viagem, chamado `criarEvento` .

```
public String criarEvento(Viagem viagem) { }
```

Criaremos um novo `Event` e colocamos a sua descrição como sendo o destino da viagem. É obrigatório informar uma lista de participantes identificados por meio do e-mail. No nosso caso, o participante é o próprio usuário; então informamos o nome da conta (que é um e-mail do Google) e incluímos no `Event` . Em seguida, criamos as datas de início e fim do evento baseados no período da viagem, incluindo as informações de fuso horário, obrigatórios para o Calendar.

```
public String criarEvento(Viagem viagem) {
    Event evento = new Event();
    evento.setSummary(viagem.getDestino());

    List<EventAttendee> participantes =
        Arrays.asList((
            new EventAttendee().setEmail(nomeConta)));
```

```
    evento.setAttendees(participantes);

    DateTime inicio = new DateTime(viagem.getDataChegada(),
                                TimeZone.getDefault());

    DateTime fim = new DateTime(viagem.getDataSaida(),
                              TimeZone.getDefault());

    evento.setStart(new EventDateTime().setDateTime(inicio));
    evento.setEnd(new EventDateTime().setDateTime(fim));

}
```

Invocamos o método de acesso à coleção `events` e nela inserimos o novo evento, especificando o calendário por meio do nome da conta. Para realizar de fato a requisição para o serviço, devemos invocar o método `execute`.

Em caso de sucesso, esse método retorna um `Event` que possui um identificador único, que utilizamos como valor de retorno. Devemos usar esse identificador posteriormente se quisermos editar ou remover o evento. Por tratar-se de uma requisição remota, é possível que uma `IOException` seja lançada e, neste caso, relançamos a exceção.

```
public String criarEvento(Viagem viagem) {
    Event evento = new Event();
    evento.setSummary(viagem.getDestino());

    List<EventAttendee> participantes =
        Arrays.asList((
            new EventAttendee().setEmail(nomeConta)));

    evento.setAttendees(participantes);

    DateTime inicio = new DateTime(viagem.getDataChegada(),
                                TimeZone.getDefault());

    DateTime fim = new DateTime(viagem.getDataSaida(),
                              TimeZone.getDefault());

    evento.setStart(new EventDateTime().setDateTime(inicio));
    evento.setEnd(new EventDateTime().setDateTime(fim));
```

```
    try {
        Event eventoCriado = calendar.events()
                                .insert(nomeConta, evento)
                                .execute();
        return eventoCriado.getId();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Já estamos quase lá. Só falta alterar a `ViagemActivity` para utilizar o `CalendarService`, e teremos a inclusão de eventos na agenda do usuário concluída.

No método `onCreate` da `ViagemActivity`, instanciaremos um `CalendarService` recuperando o nome da conta e o *token* de acesso das preferências do aplicativo:

```
// novo atributo
private CalendarService calendarService;

@Override
protected void onCreate(Bundle savedInstanceState) {
    //códigos existentes
    calendarService = criarCalendarService();
}
private CalendarService criarCalendarService() {
    SharedPreferences preferencias =
            getSharedPreferences(PREFERENCIAS, MODE_PRIVATE);
    String nomeConta = preferencias.getString(NOME_CONTA, null);
    String tokenAcesso =
        preferencias.getString(TOKEN_ACESSO, null);

    return new CalendarService(nomeConta, tokenAcesso);
}
```

Como inserir um novo evento na agenda do usuário é uma operação demorada, pois envolve acesso à rede, implementaremos uma `AsyncTask`, para executar essa operação em segundo plano. No capítulo anterior, já aprendemos como fazê-lo.

No método `salvarViagem`, logo após a gravação no banco de dados, executaremos uma `AsyncTask` para inserir o evento na

agenda do usuário.

```
public void salvarViagem(View view){
    // códigos existentes
    if(id == -1) {
        resultado = dao.inserir(viagem);
        new Task().execute(viagem);
    } else {
        resultado = dao.atualizar(viagem);
    }
    // códigos existentes
}

private class Task extends AsyncTask<Viagem, Void, Void> {
    @Override
    protected Void doInBackground(Viagem... viagens) {
        Viagem viagem = viagens[0];
        calendarService.criarEvento(viagem);
        return null;
    }
}
```

Sem mais delongas, inicie o aplicativo, cadastre uma nova viagem e confira o evento criado na sua agenda do Google Calendar! Para realizar as operações de atualização e remoção de eventos do Google Calendar, é necessário referenciar o `id` do evento desejado que é retornado quando o mesmo é criado. A partir dele, poderíamos implementar os métodos de remoção e atualização no `CalendarService` , invocando a API cliente da seguinte forma:

```
// para remover
calendar.events().delete(nomeConta, "eventId").execute();

// para atualizar
service.events().update(nomeConta, "eventId", eventoAtualizado)
        .execute();
```

7.10 CONCLUSÃO

Neste capítulo, realizamos uma integração para incluir um novo evento na agenda do usuário no Google Calendar, quando uma nova viagem for criada no aplicativo BoaViagem. Além disso,

também utilizamos uma conta registrada no dispositivo para realizar a autenticação dos usuários; aprendemos um pouco sobre o protocolo OAuth 2.0 e como aproveitar a Google Client Library para realizar a autorização de um aplicativo; e também efetuamos chamadas remotas ao serviço Google Calendar.

O conhecimento adquirido neste capítulo pode ser aplicado para o uso de qualquer outra API do Google. Aproveite-as!

CAPÍTULO 8

EXPLORE OS RECURSOS DE HARDWARE

Uma das características que tornam a plataforma Android amigável ao desenvolvedor é a existência de APIs para facilitar a manipulação dos recursos de hardware. Quando bem usados, eles incrementam as funcionalidades das aplicações e melhoram a experiência do usuário.

Neste capítulo, veremos como utilizar a câmera do dispositivo para capturar e exibir imagens e vídeos. Além disso, por meio do `MediaPlayer`, faremos a reprodução de músicas e vídeos, e implementaremos um exemplo usando o GPS, para obter a localização do aparelho e exibi-la em um mapa.

Para começar, crie um novo projeto Android para testar os códigos apresentados. Chame-o de `Hardware` e utilize o pacote `br.com.casadocodigo.hardware`. Nas seções seguintes, veremos códigos para trabalhar com alguns dos recursos de hardware disponíveis. É recomendado usar um aparelho Android em vez do emulador, pois, dessa forma, será mais rápido e mais fácil verificar o resultado das implementações.

8.1 CAPTURE FOTOS COM SEU APARELHO

Um dos recursos que mais se popularizou entre os telefones celulares, antes mesmo da existência dos *smartphones*, foi a câmera

fotográfica embutida. Como uma imagem publicada em uma rede social vale mais do que 140 caracteres, aplicativos, como Instagram, Skitch e vários outros, exploram esse recurso para criar novas formas de interação.

Nesta seção, o objetivo é conhecer como usar a câmera do dispositivo em um aplicativo Android. Em nosso exemplo, vamos requisitar ao próprio aplicativo da câmera para que a foto seja capturada, utilizando uma `Intent` . Esta é uma forma simples e prática de incluir essa funcionalidade no seu aplicativo.

As imagens capturadas devem ser armazenadas no cartão de memória para evitar que o espaço de armazenamento interno, que é mais limitado e usado pelo sistema operacional, não seja comprometido. Além disso, o usuário pode retirar o cartão do dispositivo e utilizá-lo em seu computador para recuperar as fotos facilmente.

Por padrão, existem dois diretórios que podemos usar para salvar as imagens capturadas. O primeiro é o `getExternalStoragePublicDirectory` da classe `Environment` , que retorna um diretório compartilhado por todas as aplicações, e é o recomendado para o armazenamento de fotos e vídeos. Como ele não está associado ao aplicativo que gravou a imagem, quando este é desinstalado, os dados não são perdidos.

O outro diretório é obtido por meio de `Context.getExternalFilesDir` , que retorna um diretório associado ao aplicativo. As imagens salvas nele são removidas quando o aplicativo é desinstalado. A escolha de qual utilizar depende do propósito do aplicativo e de qual comportamento é o mais adequado em caso de desinstalação.

Para iniciar os testes com a câmera, crie uma nova *activity* com o nome de `CameraActivity` . Nela implementaremos as

funcionalidades de captura e visualização da imagem. Seu `layout` (`câmera.xml`) terá dois botões para disparar as funcionalidades citadas:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/capturar_imagem"
        android:onClick="capturarImagem"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/visualizar_imagem"
        android:onClick="visualizarImagem"/>

</LinearLayout>
```

Na `CameraActivity` faremos a chamada ao aplicativo da câmera por meio de uma `Intent` . No entanto, precisaremos obter uma resposta desse aplicativo para saber se a captura foi bem sucedida ou não. Nessas situações, podemos iniciar uma atividade usando o método `startActivityForResult` . Quando a operação solicitada for finalizada, o método `onActivityResult` da atividade que fez a solicitação é invocado. O código inicial da `CameraActivity` é o seguinte:

```
public class CameraActivity extends Activity {
    private static final int CAPTURAR_IMAGEM = 1;
    private Uri uri;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.camera);
    }
```

```
    @Override
    protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    }

    public void capturarImagem(View v){
    }

    public void visualizarImagem(View v){
    }
}
```

Para realizar a captura da imagem, precisaremos criar uma `Intent` e passar como extra a `Uri`, informando o nome e local de armazenamento da imagem. Utilizaremos o diretório público, que é o recomendado.

Recuperamos o diretório e o escolhemos para armazenar nossas imagens. Em seguida, geramos o nome da imagem com base no diretório obtido, usando a informação de tempo do sistema em milissegundos para termos um nome único. A extensão do arquivo é definida como `.jpg`. Assim, vamos recuperar o objeto `Uri` com o caminho:

```
public void capturarImagem(View v) {
    File diretorio = Environment
                    .getExternalStoragePublicDirectory(
                            Environment.DIRECTORY_PICTURES);

    String nomeImagem = diretorio.getPath() + "/" +
                        System.currentTimeMillis() +
                        ".jpg";

    uri = Uri.fromFile(new File(nomeImagem));
}
```

Precisamos definir a `Intent` e utilizá-la para iniciar uma nova atividade que deve retornar um resultado. Para o método `startActivityForResult`, é necessário informar um código para identificar a solicitação. Utilizamos a constante `CAPTURAR_IMAGEM` com esta finalidade. Isso é necessário, pois podemos iniciar várias atividades distintas que devem retornar resultados. Em nosso

exemplo mais à frente, usaremos o método `startActivityForResult` com uma `Intent`, para capturar vídeos.

```
public void capturarImagem(View v) {
    // Recupera o arquivo e a URI

    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    intent.putExtra(MediaStore.EXTRA_OUTPUT, uri);

    startActivityForResult(intent, CAPTURAR_IMAGEM);
}
```

Quando o aplicativo da câmera tirar a foto e o usuário confirmar a captura, a *activity* da câmera será finalizada com um resultado, e o método `onActivityResult` da `CameraActivity` será invocado. Nesse método, faremos uma checagem para saber se o resultado é para a solicitação que foi feita com o código `CAPTURAR_IMAGEM`.

Em seguida, avaliaremos se o resultado informado pela câmera se refere ao sucesso ou falha na captura da imagem. Em caso de sucesso, a imagem deve ser adicionada à galeria de fotos do Android. A implementação do `onActivityResult` é a seguinte:

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    if (requestCode == CAPTURAR_IMAGEM) {
        if (resultCode == RESULT_OK) {
            mostrarMensagem("Imagem capturada!");
            adicionarNaGaleria();
        } else {
            mostrarMensagem("Imagem não capturada!");
        }
    }
}
private void mostrarMensagem(String msg){
    Toast.makeText(this, msg,
                Toast.LENGTH_LONG)
                .show();
}
```

Por mais que tenhamos armazenado a imagem em um diretório

compartilhado e acessível pela galeria de fotos do Android, ela não é adicionada lá automaticamente. É necessário que façamos a sua inclusão. Para isso, basta disparar um `broadcast` para notificar que a foto deve ser incluída na galeria usando a `Uri` da imagem. O código para realizar essa operação é o seguinte:

```
private void adicionarNaGaleria() {
    Intent intent = new Intent(
                    Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    intent.setData(uri);
    this.sendBroadcast(intent);
}
```

Para exibir a imagem capturada, utilizaremos uma `Intent` para invocar a galeria de imagens do Android. No método `visualizarImagem`, passaremos a `Uri` e o *MIME type* da imagem na `Intent` para iniciar a *activity*. Veja como fica o código:

```
public void visualizarImagem(View v){
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setDataAndType(uri, "image/jpeg");
    startActivity(intent);
}
```

Assim, temos as funcionalidades de capturar e visualizar imagens, usando `Intents` prontas! Para iniciar a `CameraActivity`, altere o `main.xml` para incluir um botão que disparará essa opção.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/camera"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="escolherOpcao"
        android:text="@string/testar_camera" />
```

```
</LinearLayout>
```

Na `HardwareActivity`, implemente o método `escolherOpcao` conforme o código a seguir, para abrir a `Intent` para a câmera.

```
public void escolherOpcao(View view){
    if(view.getId() == R.id.camera){
        Intent intent = new Intent(this, CameraActivity.class);
        startActivity(intent);
    }
}
```

Pronto, agora é só executar a aplicação e testar a captura de fotos!

Nas situações nas quais a câmera é essencial para o funcionamento do aplicativo, podemos incluir essa obrigatoriedade no `AndroidManifest.xml`. Dessa forma, dispositivos que não possuem câmera não poderão instalar o aplicativo. Também é possível informar que o aplicativo utiliza a câmera, mas que ela não é obrigatória.

Para checar a existência da câmera em tempo de execução, podemos fazer o seguinte:

```
public void capturarImagem(View v){
    boolean temCamera = getPackageManager()
        .hasSystemFeature(PackageManager.FEATURE_CAMERA);
    if(temCamera){
        //códigos implementados para a captura de imagens
    }
}
```

A seguir, veja a forma de declarar o uso da câmera no manifesto como requisito para a aplicação. Repare que, como não estamos usando diretamente o hardware da câmera, não é necessário adicionar nenhuma permissão.

```
<!-- Câmera é obrigatória -->
<uses-feature android:name="android.hardware.camera"/>
<!-- Câmera não é obrigatória -->
```

```
<uses-feature android:name="android.hardware.camera"
    android:required="false"/>
```

8.2 GRAVE VÍDEOS

Outro recurso que podemos explorar em nossos aplicativos é a gravação de vídeos. Também é possível utilizar `Intent` para requisitar a gravação para a câmera do dispositivo, assim como é feito para a captura de imagens.

Além da possibilidade de informar onde o vídeo deve ser armazenado, podemos incluir outras informações extras na `Intent` para configurar sua captura:

- `MediaStore.EXTRA_DURATION_LIMIT` — configura a duração limite da captura de vídeo em segundos.
- `MediaStore.EXTRA_VIDEO_QUALITY` — determina em qual qualidade o vídeo deve ser capturado. Para vídeos em baixa qualidade, devemos informar o valor `0` e para os vídeos em alta qualidade, o valor `1` .
- `MediaStore.EXTRA_SIZE_LIMIT` — define o tamanho limite em *bytes* para o vídeo que está sendo capturado.

Para testar a gravação de vídeos, na `CameraActivity` criaremos um novo método que usará o aplicativo da câmera para gravar em alta qualidade, com duração de 5 segundos. Não informaremos nada para o extra `MediaStore.EXTRA_OUTPUT` ; o que fará com que o vídeo gravado seja armazenado no diretório padrão e com um nome gerado automaticamente. Veja o código para iniciar a `Intent` :

```
// nova constante para identificar a requisição
private static final int CAPTURAR_VIDEO = 2;

public void capturarVideo(View v){
    Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);
```

```
    intent.putExtra(MediaStore.EXTRA_DURATION_LIMIT, 5);
    startActivityForResult(intent, CAPTURAR_VIDEO);
}
```

No método `onStartActivity`, teremos que verificar se o resultado obtido se refere à captura de vídeo, e utilizaremos a `Intent` recebida como parâmetro para saber o nome do arquivo e o local onde ele foi armazenado. Veja:

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    if (requestCode == CAPTURAR_IMAGEM) {
        if (resultCode == RESULT_OK) {
            mostrarMensagem("Imagem capturada!");
            adicionarNaGaleria();
        } else {
            mostrarMensagem("Imagem não capturada!");
        }
    } else if(requestCode == CAPTURAR_VIDEO) {
        if (resultCode == RESULT_OK) {
            String msg = "Vídeo gravado em  " +
                            data.getDataString();
            mostrarMensagem(msg);
            uri = data.getData();
        } else {
            mostrarMensagem("Vídeo não gravado");
        }
    }
}
```

Para exibir o vídeo, criaremos uma `Intent` utilizando a `Uri` recuperada de `Intent.getData()`, que representa o arquivo que desejamos visualizar.

```
public void visualizarVideo(View v){
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setDataAndType(uri, "video/mp4");
    startActivity(intent);
}
```

Para testar essas novas funcionalidades, acrescente em `camera.xml` dois `Buttons` para chamar os métodos de capturar e visualizar o vídeo.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ... >
    <!-- botoes existentes -- >
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="capturarVideo"
        android:text="@string/capturar_video" />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="visualizarVideo"
        android:text="@string/visualizar_video" />

</LinearLayout>
```

8.3 EXECUTE VÍDEOS E MÚSICAS

Nesta seção, vamos implementar algumas formas de executar vídeos e músicas em uma aplicação Android, usando basicamente a classe `MediaPlayer` e uma `VideoView` para exibir os vídeos. Comece criando uma nova atividade com o nome de `MediaPlayerActivity` e um arquivo de layout `media_player.xml`, que contará com alguns botões para acionar as diferentes formas de execução de músicas e de vídeo que vamos implementar, e também controlará a sua execução. A definição será a seguinte:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="executarMusicaArquivo"
        android:text="@string/executar_musica_arquivo" />

    <Button
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:onClick="executarMusicaUrl"
        android:text="@string/executar_musica_url" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="executarVideo"
        android:text="@string/executar_video" />

    <TableRow>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="parar"
            android:text="@string/parar" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="executar"
            android:text="@string/executar" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="pausar"
            android:text="@string/pausar" />
    </TableRow>

</TableLayout>
```

O primeiro exemplo consiste em executar um arquivo `.mp3` , existente dentro da aplicação. Escolha uma das suas músicas preferidas e a inclua na pasta `res/raw` do projeto (crie a pasta `raw` , se necessário). É importante que o nome do arquivo não contenha caracteres especiais; por isso, neste exemplo, utilizaremos um arquivo chamado `musica.mp3` .

Na classe `MediaPlayerActivity` , criaremos um novo `MediaPlayer` , carregando este arquivo de música. Em seguida, usaremos os métodos disponíveis no `MediaPlayer` para controlar sua execução. Veja o código a seguir:

```
public class MediaPlayerActivity extends Activity{

    private MediaPlayer player;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.media_player);
    }

    public void executarMusicaArquivo(View v) {
        player = MediaPlayer.create(this, R.raw.musica);
        player.start();
    }
}
```

Utilizamos o método utilitário `MediaPlayer.create`, passando o contexto e o identificador do arquivo de música para criar um novo `MediaPlayer`. Para iniciar a execução, invocamos o método `start`. No método `create`, o arquivo foi carregado e o `MediaPlayer` *preparado* para a execução. Os controles serão feitos usando os métodos respectivos, presentes no `MediaPlayer`:

```
public void executar(View v) {
    if(!player.isPlaying()) {
        player.start();
    }
}

public void pausar(View v) {
    if(player.isPlaying()) {
        player.pause();
    }
}

public void parar(View v) {
    if(player.isPlaying()) {
        player.stop();
    }
}
```

O método `start` pode ser utilizado tanto para iniciar a execução pela primeira vez quanto para retomar quando esta estiver em pausa. No entanto, se a reprodução for interrompida com o

`stop`, só é possível iniciá-la novamente após o `MediaPlayer` ser *preparado*; ou seja, o método `prepare` deve ser invocado.

Outra questão importante é que devemos sempre liberar o `MediaPlayer` quando ele não for mais necessário. Isso contribuirá para a liberação da memória e também dos *codecs* usados.

Podemos sobrescrever o método `onStop` da `MediaPlayerActivity` para tratar essa questão:

```
@Override
protected void onStop() {
    super.onStop();
    liberarPlayer();
}
private void liberarPlayer() {
    if(player != null){
        player.release();
    }
}
```

No próximo exemplo, em vez de carregar um arquivo `.mp3` existente na aplicação, faremos o seu carregamento a partir de uma URL. Esse cenário requer uma atenção especial, pois envolve acesso à Internet, o que pode tornar a operação demorada.

Já vimos que, nesses casos, devemos realizar a operação de forma assíncrona para não bloquear a UI. Como esta é uma situação recorrente, o `MediaPlayer` já disponibiliza um método de preparação assíncrona.

Precisamos implementar a interface `OnPreparedListener` que possui um método chamado `onPrepared`, que será invocado no término da preparação assíncrona. Nesse método, apenas chamaremos o `start` para iniciar a reprodução. No método `executarMusicaUrl`, liberamos a instância anterior do *player*, que pode ter sido criada para executar um arquivo da aplicação, e criamos um novo `MediaPlayer`.

```
public class MediaPlayerActivity extends Activity
        implements OnPreparedListener {

    //códigos existentes

    public void executarMusicaUrl(View v) {
        liberarPlayer();
        player = new MediaPlayer();
    }

    @Override
    public void onPrepared(MediaPlayer mp) {
        player.start();
    }
}
```

Agora, precisamos criar uma `Uri` a partir de uma URL qualquer, que referencia o arquivo `.mp3` desejado. Em seguida, a atribuímos como `datasource` do `MediaPlayer`, além de informar que ela deverá ser executada. Por fim, atribuímos a própria *activity* como o *listener* da preparação assíncrona que é invocada.

```
public class MediaPlayerActivity extends Activity
        implements OnPreparedListener {

    //códigos existentes

    public void executarMusicaUrl(View v) {
        liberarPlayer();
        player = new MediaPlayer();
        try {
            Uri uri =
                Uri.parse("http://<alguma-url>/musica.mp3");
            player.setDataSource(this, uri);
            player.setAudioStreamType(
                AudioManager.STREAM_MUSIC);
            player.setOnPreparedListener(this);
            player.prepareAsync();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void onPrepared(MediaPlayer mp) {
        player.start();
```

```
    }
}
```

Com isto, temos a reprodução de músicas da Internet e também aquelas existentes no aplicativo implementadas! Antes de testar, inclua a permissão de acesso à Internet no `AndroidManifest.xml` :

```
<uses-permission android:name="android.permission.INTERNET" />
```

> **Formatos suportados**
>
> O Android suporta diversos formatos de áudio e vídeo, dentre eles arquivos `.mp3` , `.mp4` e `.3gp` . Para uma relação completa dos formatos suportados, consulte http://developer.android.com/guide/appendix/media-formats.html.

Agora precisamos exibir um vídeo armazenado no aplicativo. Inclua na pasta `res/raw` um arquivo de vídeo com um dos formatos suportados. Usaremos um arquivo com nome de `video.mp4` . Ao contrário da reprodução de um arquivo de música, em que manipulamos o `MediaPlayer` diretamente, agora vamos utilizar uma `view` que, além de realizar a exibição do vídeo propriamente dito, também oferece os controles de execução. No arquivo `media_player.xml` , inclua uma `VideoView` :

```
<TableLayout ...>
<!-- códigos existentes -->
    <VideoView android:id="@+id/videoView" />
</TableLayout>
```

No método `onCreate` da `MediaPlayerActivity` , obteremos uma referência para a `VideoView` que utilizaremos na implementação do método `executarVideo` . Basicamente, o que vamos fazer é criar uma `Uri` para o vídeo armazenado no

dispositivo; atribuí-la para a `VideoView`; adicionar os controles de execução, que são implementados pela classe `MediaController`; e, por fim, iniciar a reprodução, invocando o método `VideoView.start`.

```
public class MediaPlayerActivity extends Activity
        implements OnPreparedListener{

    private MediaPlayer player;
    private VideoView videoView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.media_player);
        videoView = (VideoView) findViewById(R.id.videoView);
    }

    public void executarVideo(View v) {
        Uri uri = Uri.parse("android.resource://"
            + getPackageName() + "/" + R.raw.video);

        videoView.setVideoURI(uri);
        MediaController mc = new MediaController(this);
        videoView.setMediaController(mc);
        videoView.start();
    }
}
```

A `Uri` informada para a `VideoView` pode se referir tanto a um arquivo local como a um arquivo remoto. A `VideoView` já trata a preparação assíncrona do *player* e também o libera quando não está mais em uso. Para abrir esta nova `activity`, acrescente um botão em `main.xml` e implemente os códigos necessários no método `escolherOpcao` da `HardwareActivity`:

```
<!-- main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout...>
    <!-- botoes existentes -->
    <Button
        android:id="@+id/mediaplayer"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:onClick="escolherOpcao"
        android:text="@string/testar_media_player" />
</LinearLayout>

public void escolherOpcao(View view){
    if(view.getId() == R.id.mediaplayer){
        Intent intent =
            new Intent(this, MediaPlayerActivity.class);
        startActivity(intent);
    }
}
```

Execute a aplicação e experimente a reprodução de vídeos.

8.4 DETERMINE A LOCALIZAÇÃO ATRAVÉS DO GPS E DA REDE

Grande parte dos smartphones e tablets Android disponíveis no mercado contam com um sistema de posicionamento global, o GPS. Através dele, é possível determinar a localização do dispositivo com boa precisão. No entanto, para funcionar adequadamente, é necessário que o dispositivo esteja em um ambiente aberto para facilitar a comunicação com os satélites.

O processo de localizar e conectar aos satélites pode ser demorado, o que aumenta o tempo de espera por informações de localização.

Outra forma de se obter a localização é por meio do *Network Location Provider*, que utiliza os sinais da rede de celular e Wi-Fi para determinar a localização do usuário. Apesar de ser menos precisa, essa forma consome menos bateria e obtém resultados de localização com mais rapidez, além de funcionar tanto em ambientes abertos quanto em recintos fechados.

A partir dessas informações de localização, podemos, por exemplo, desenvolver um aplicativo que apresenta em um mapa os pontos de interesse próximos ao usuário, como supermercados,

farmácias etc. A localização do usuário passa a ser relevante.

Vamos usar o GPS e a rede para obter as coordenadas de latitude e longitude, e apresentar essa localização em um mapa do Google. Continuaremos utilizando o mesmo projeto `Hardware`.

Para acessar informações de localização, a plataforma Android disponibiliza o `LocationManager`. Por meio dessa classe, podemos registrar um `LocatitonListener` para receber as atualizações de localização tanto de um `GPS_PROVIDER` como de um `NETWORK_PROVIDER`. Além de receber dados de localização, o `listener` é notificado quando algum provedor é habilitado ou desabilitado pelo usuário e também quando este muda de estado (fora de serviço, disponível, temporariamente indisponível ou outro).

Conforme o provedor de localização (GPS, rede ou ambos) que o aplicativo utilizará, é necessário declarar as permissões adequadas no `AndroidManifest.xml`. Quando o aplicativo usar apenas o provedor de rede, então a permissão que deve ser declarada é a seguinte:

```
<uses-permission
    android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Já no caso que o GPS será utilizado, a permissão necessária é a seguinte:

```
<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Caso o aplicativo use os dois provedores de localização, basta declarar a permissão `android.permission.ACCESS_FINE_LOCATION`, que, além de autorizar o uso do GPS, também permite a utilização do *Network Location Provider*. Inclua essa permissão no manifesto do projeto.

Para iniciar a implementação, crie uma nova classe com o nome de `LocalizacaoActivity` e um novo arquivo de layout para ela com o nome de `localizacao.xml` . Inicialmente, nesta tela apresentaremos as coordenadas de latitude e longitude, bem com o *provider* responsável por essas informações. Veja a definição do layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView android:id="@+id/provedor"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView android:id="@+id/latitude"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView android:id="@+id/longitude"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Na `LocalizacaoActivity` registraremos um `listener` para receber as atualizações de localização e exibiremos essas informações nos `TextViews` .

Precisaremos obter o `LocationManager` , que é responsável por gerenciar os provedores de localização. Depois, declaramos o nosso `listener` , que veremos na sequência, e o registramos para receber as atualizações oriundas do `NETWORK_PROVIDER` e do `GPS_PROVIDER` .

Para registrar um `listener` , usamos o método `requestLocationUpdates` informando o provedor desejado, o intervalo de tempo em milissegundos e a distância em metros entre as atualizações. Ao configurar estes dois parâmetros como `0` , isso

indicará que as atualizações de localização devem ser realizadas o mais frequentemente possível.

```
public class LocalizacaoActivity extends Activity {
    private LocationManager locationManager;
    private TextView latitude, longitude, provedor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.localizacao);

        latitude = (TextView) findViewById(R.id.latitude);
        longitude = (TextView) findViewById(R.id.longitude);
        provedor = (TextView) findViewById(R.id.provedor);

        locationManager = (LocationManager)
                this.getSystemService(Context.LOCATION_SERVICE);

        Listener listener = new Listener();

        long tempoAtualizacao = 0;
        float distancia = 0;

        locationManager.requestLocationUpdates(
                LocationManager.NETWORK_PROVIDER,
                tempoAtualizacao, distancia, listener);

        locationManager.requestLocationUpdates(
                LocationManager.GPS_PROVIDER,
                tempoAtualizacao, distancia, listener);
    }
}
```

Repare que registramos o `listener` para receber atualizações tanto do `NETWORK_PROVIDER` como do `GPS_PROVIDER` .

Agora, precisamos implementar um `LocationListener` que receberá as informações de localização e atualizará a tela com esses dados. Para isso, criamos uma classe privada chamada `Listener` .

Quando a localização obtida por algum dos provedores for alterada, o método `onLocationChanged` é invocado com o nova localização sendo enviada como parâmetro. A classe `Location`

fornece métodos para recuperar as informações de latitude, longitude e o provedor de origem. Usamos esses métodos para atualizar o valor dos `TextViews` .

```
private class Listener implements LocationListener{

    @Override
    public void onLocationChanged(Location location) {
        String latitudeStr =
            String.valueOf(location.getLatitude());
        String longitudeStr =
            String.valueOf(location.getLongitude());

        provedor.setText(location.getProvider());
        latitude.setText(latitudeStr);
        longitude.setText(longitudeStr);
    }

    @Override
    public void onStatusChanged(String provider, int status,
        Bundle extras) {}

    @Override
    public void onProviderEnabled(String provider) {}

    @Override
    public void onProviderDisabled(String provider) {}
}
```

Para executar esse exemplo e realizar os testes de localização, é recomendável que seja utilizado um dispositivo Android, em vez do emulador.

Caso não tenha um aparelho disponível, é possível usar o emulador e simular as localizações de GPS. Para isso, no Android Studio vá em `Tools > Android > Android Device Monitor` . Uma nova aplicação será aberta; prossiga para a aba `Emulator Control` , como mostra a figura a seguir. Por meio dela, poderemos enviar informações de localização simulada para o emulador.

Figura 8.1: Emulator control

Inclua a `LocalizacaoActivity` no manifesto, crie um botão no `main.xml` e altere o método `escolherOpcao` da `HardwareActivity` para iniciar a nova atividade.

```
<!-- main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout...>
    <!-- botoes existentes -->
    <Button
        android:id="@+id/localizacao"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="escolherOpcao"
        android:text="@string/testar_localizacao" />

</LinearLayout>

public void escolherOpcao(View view){
    //códigos existentes
    if(view.getId() == R.id.localizacao){
```

```
        Intent intent =
            new Intent(this, LocalizacaoActivity.class);
        startActivity(intent);
    }
}
```

Agora sim! Execute a aplicação e veja a sua localização. Se estiver utilizando um dispositivo para testar, inicialmente a localização obtida será da rede e, posteriormente, do GPS, caso algum sinal seja obtido. A figura a seguir mostra como ficou a aplicação usando a localização.

Figura 8.2: Emulator control usando a localização

O próximo passo é utilizar essas informações de latitude e longitude para exibir a localização em um mapa. Para fazer isso, podemos usar o add-on Google APIs, que conta com uma biblioteca para usar o Google Maps no Android, por meio de uma `MapView` . No entanto, essa abordagem requer uma série de passos de preparação que só é viável quando o aplicativo necessita manipular e interagir intensamente com mapas.

No nosso exemplo, no qual queremos apenas exibir a localização, podemos utilizar a Google Static Maps API, que permite a recuperação de um mapa em formato de imagem, construído a partir de alguns parâmetros informados em uma URL. Resumindo, o que faremos é uma chamada para uma URL, informando as coordenadas da localização como parâmetro e receberemos como resposta uma imagem que é o mapa propriamente dito.

Tanto a requisição para o serviço do Google quanto a exibição da imagem serão feitas usando uma `view` do Android, chamada de `WebView`. Ela permite a exibição de páginas da web que utiliza o mesmo *engine* do navegador disponível no Android. Ou seja, quando incluímos uma `WebView` em nosso aplicativo, temos praticamente todos os recursos do navegador padrão.

Para carregar páginas da web em uma `WebView`, é necessário incluir a seguinte permissão no manifesto:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Em seguida, adicione uma `WebView` no arquivo de layout `localizacao.xml`, dessa forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout...>
    <!-- textviews existentes -->
<WebView
        android:id="@+id/mapa"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Na `LocalizacaoActivity`, criaremos dois novos atributos: um para a `WebView` e o outro para armazenar a URL base, para o serviço de mapas estáticos. No método `onCreate`, recuperamos a `WebView` para usá-la posteriormente.

```
private String urlBase = "http://maps.googleapis.com/maps/api" +
    "/staticmap?size=400x400&sensor=true" +
    " &markers=color:red|%s,%s";
private WebView mapa;
@Override
protected void onCreate(Bundle savedInstanceState) {
    // códigos existentes
    mapa = (WebView) findViewById(R.id.mapa);
}
```

A URL do serviço de mapas já está configurada com alguns parâmetros, que são o tamanho da imagem que deve ser retornada, uma `flag` indicando que as coordenadas foram obtidas de um

sensor e as configurações do marcador da localização.

O marcador terá a cor vermelha e será posicionado nas coordenadas de latitude e longitude, que serão incluídas no lugar dos `s` . Estes são alguns dos parâmetros disponíveis na Google Static Maps API — para saber mais, visite https://developers.google.com/maps/documentation/staticmaps/.

Para exibir o mapa na `WebView` , basta montar a URL com as coordenadas e carregá-la com o método `loadUrl` . Veja:

```
@Override
public void onLocationChanged(Location location) {
    // códigos existentes
    String url =
        String.format(urlBase, latitudeStr, longitudeStr);
    mapa.loadUrl(url);
}
```

Dessa forma, sempre que a localização mudar, um novo mapa será exibido. A figura adiante demonstra como o mapa será exibido na aplicação de exemplo. É importante ressaltar que, neste caso, estamos utilizando uma configuração para receber as atualizações de localização o mais frequentemente possível. Em cenários reais, isso não é recomendado, pois aumenta o consumo da bateria.

Figura 8.3: Localização na WebView

8.5 CONCLUSÃO

Neste capítulo, vimos como utilizar a câmera do dispositivo para incrementar nossas aplicações fazendo uso de imagens e vídeos. Também usamos o `MediaPlayer` para reproduzir músicas e vídeos tanto de arquivos armazenados no dispositivo como de arquivos da web.

Através do GPS, recuperamos as coordenadas de latitude e longitude e exibimos a localização obtida utilizando um mapa gerado pela Google Static Maps API.

Para facilitar o uso da Static Maps API, que é um serviço REST que retorna uma imagem, lançamos mão de uma `WebView` que permite renderizar páginas da web. Com estes conhecimentos é

possível incrementar as funcionalidades dos seus aplicativos.

Que tal agora incluir no BoaViagem as coordenadas geográficas representando o local onde o gasto foi realizado? Bons códigos!

CAPÍTULO 9

SUPORTE TABLETS E OUTROS DISPOSITIVOS

A versão 4 do Android unificou a plataforma para a sua utilização tanto em tablets como em smartphones. Antes disso, tínhamos a versão 3.x projetada, especificamente, para tablets e a versão 2.x amplamente usada em smartphones e em alguns modelos de tablet.

A versão 3.0 introduziu novos recursos de UI, como a `ActionBar` e os `Fragments`, que são componentes de interface gráfica modulares com ciclo de vida próprio, criados para facilitar o desenvolvimento de aplicativos com layouts diferentes para cada tipo de dispositivo. Para trazer alguns desses recursos para as versões anteriores do Android, o Google lançou um pacote de compatibilidade que permite a utilização de `Loaders` e `Fragments`, inclusive na sua versão 1.6.

Neste capítulo, veremos como usar o pacote de compatibilidade para utilizar os recursos mais recentes da plataforma, bem como conhecer as formas de suportar diversas versões do Android e vários tamanhos de tela, incluindo tablets.

9.1 PREPARE O SEU AMBIENTE

Para implementar os exemplos deste capítulo, será necessário fazer o download do pacote de compatibilidade e também criar um

novo AVD para emular um tablet de 10 polegadas.

O pacote de compatibilidade é obtido através do *Android SDK Manager*. No Android Studio, acesse o menu `Tools > Android > SDK Manager`. Na janela apresentada, selecione a opção `Android Support Library`, como mostra a figura a seguir. Aproveite e também atualize as ferramentas do SDK para as versões mais recentes, se estiverem disponíveis.

Figura 9.1: Adicionado bibliotecas de suporte

Depois que os downloads acabarem, abra o `AVD Manager` pelo menu `Tools > Android` da IDE e crie um novo AVD com os dados ilustrados nas figuras a seguir, para emular um tablet baseado no Nexus 10".

Figura 9.2: Criando novo AVD

Figura 9.3: Emulando um tablet

Para utilizar a biblioteca de suporte no projeto, é necessário adicioná-la como dependência do Gradle. Para isso, no arquivo `build.gradle` do módulo referente à sua aplicação, adicione a seguinte declaração, caso o Android Studio ainda não tenha colocado para você na criação do projeto:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.0'
}
```

9.2 SUPORTE VÁRIAS VERSÕES DO ANDROID

O Android pode ser executado em uma quantidade bastante diversificada de dispositivos. Por um lado, isso aumenta a gama de usuários; por outro, torna necessário que os desenvolvedores levem em consideração qual público/dispositivos atenderão com seus aplicativos.

Em julho de 2015, as versões 4.x do Android já representam a grande maioria da base instalada de dispositivos. A figura a seguir mostra a participação de cada versão com base nos dispositivos que acessam o Google Play.

Version	Codename	API	Distribution
2.2	Froyo	8	0.3%
2.3.3 - 2.3.7	Gingerbread	10	5.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.1%
4.1.x	Jelly Bean	16	14.7%
4.2.x		17	17.5%
4.3		18	5.2%
4.4	KitKat	19	39.2%
5.0	Lollipop	21	11.6%
5.1		22	0.8%

Data collected during a 7-day period ending on June 1, 2015.

Figura 9.4: Version share

O Google recomenda que o seu aplicativo seja compatível com cerca de 90% da base instalada. De acordo com os dados de julho de 2015, para atender essa meta, nosso aplicativo BoaViagem precisa ser compatível com a versão 2.3.3 do Android.

Manter a compatibilidade com versões anteriores exige que se abra mão de recursos disponíveis em APIs mais recentes; ou pior, manter códigos específicos para cada versão, causando duplicação de código e dificultando a manutenibilidade. Nos casos em que é necessário verificar, em tempo de execução, qual é a versão do Android, usamos as informações contidas na classe `Build`. Veja como fica um código que verifica se o aplicativo está sendo executado em uma versão superior a 1.6, para decidir como realizar determinada operação:

```
private void algumaOperacao() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.DONUT) {
        // usar api nova
    } else {
        // usar api antiga
    }
}
```

Geralmente, quando sai uma nova versão do Android, novos atributos para os XMLs também são incluídos. Isso quer dizer que teremos versões diferentes de XMLs para cada versão? Felizmente, não. O Android automaticamente ignora esses novos atributos quando o aplicativo estiver sendo executado em uma versão anterior que não os possui.

Para definir quais versões ele suporta, devemos declarar no `AndroidManifest.xml` qual é a versão mínima e a versão alvo; ou seja, a versão na qual o aplicativo foi desenvolvido, testado e sabe-se que funciona corretamente. Veja um exemplo no qual a versão mínima é a 2.2 (API 8) e a versão alvo é a 5.1 (API 22):

```
<manifest ... >
    <uses-sdk android:minSdkVersion="8"
              android:targetSdkVersion="22" />
    <!-- demais declarações -->
</manifest>
```

Como estamos trabalhando com o Gradle, também é possível (e preferível) realizar essa configuração no arquivo `build.gradle`.

Veja como:

```
defaultConfig {
        applicationId "br.com.casadocodigo.boaviagem"
        minSdkVersion 10
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
}
```

Além de restringir por versões, também podemos delimitar a instalação do aplicativo com base nas características e recursos disponíveis nos dispositivos. Podemos, por exemplo, definir que nosso aplicativo só seja instalado em um dispositivo que possua câmera e GPS.

```
<manifest ... >
    <uses-feature android:name="android.hardware.location.gps"/>
    <uses-feature android:name="android.hardware.camera" />
    <!-- demais declarações -->
</manifest>
```

Para uma relação completa de todas os filtros que podem ser aplicados para impedir que seu aplicativo seja instalado por um dispositivo que não oferece os recursos necessários, visite http://developer.android.com/guide/google/play/filters.html. Estas configurações não são obrigatórias, mas, sem dúvida, auxiliam a direcionar seu aplicativo apenas para os dispositivos que possam executá-lo corretamente.

9.3 SUPORTE DIVERSOS TAMANHOS DE TELA

Atualmente, existe uma grande variedade de dispositivos rodando a plataforma Android, desde smartphones com telas compactas, tablets de diferentes tamanhos e até mesmo Smart TVs com telas que ultrapassam 32 polegadas.

Para que o usuário tenha uma boa experiência de uso, independentemente do dispositivo que esteja utilizando, é

importante que seu aplicativo suporte vários tipos de tela. Assim, tirará proveito das telas maiores, para exibir mais conteúdo, e otimizará a visualização de informações, no caso das telas menores.

Para cada tamanho de tela que se deseja suportar, um layout diferente deve ser definido. De acordo com o dispositivo que está executando a aplicação, o Android aplica o layout adequado.

Na plataforma Android, as telas são classificadas de acordo com duas características principais: a densidade e o tamanho. A densidade refere-se à quantidade de pixels existentes em uma determinada área da tela (uma polegada), cuja unidade de medida é o `dpi` (*dots per inch*). Por simplificação, o Android define quatro densidades: baixa (`ldpi`), média (`mdpi`), alta (`hdpi`) e extra alta (`xhdpi`).

A densidade deve ser levada em consideração quando estamos definindo as dimensões dos componentes de UI e também confeccionando as imagens (`drawables`) que serão usadas no aplicativo. Supondo que temos uma imagem com tamanho 100x100 para uma tela de densidade média, para atender as demais densidades, teremos outras versões com tamanho: 75x75 para baixa, 150x150 para alta e 200x200 para extra alta.

A escala usada para determinar seu tamanho, de acordo com a densidade, é a seguinte:

- ldpi — 0.75
- mdpi — 1.0
- hdpi — 1.5
- xhdpi — 2.0

Uma vez criadas, as imagens devem ser colocadas nos diretórios de `drawables` específicos para cada densidade. Veja um exemplo:

```
BoaViagem/
```

```
res/
  drawable-xhdpi/
      configuracoes.png
  drawable-hdpi/
      configuracoes.png
  drawable-mdpi/
      configuracoes.png
  drawable-ldpi/
      configuracoes.png
```

De acordo com a densidade do dispositivo, o Android escolherá a imagem adequada em tempo de execução. Caso você não forneça imagens para cada tipo de densidade, o sistema fará por conta própria seu redimensionamento, podendo levar a perda de qualidade. No entanto, se a única imagem disponível se refere a uma densidade maior do que à disposição, o Android lançará um erro.

Já o tamanho se refere ao tamanho físico da tela, medido pelo comprimento (em polegadas) da sua diagonal. Alguns exemplos de tamanho de tela são as de **4.7"**, **7"** e **10.1"** polegadas. O Android agrupa-as em quatro tamanhos: pequena (*small*), normal (*normal*), grande (*large*) e extra grande (*extra large*).

Além disso, ainda existe a orientação da tela que pode ser retrato (*portrait*) ou paisagem (*landscape*), de acordo com o ponto de vista do usuário. O seu aplicativo deve levar em conta as mudanças de orientação do dispositivo que acontecem em tempo de execução para ajustar o layout, conforme necessário. Podemos criar layouts para tamanhos e orientações de tela específicos. Para isso, basta criar o layout desejado e colocá-lo no diretório correspondente, seguindo a convenção:

```
res/layout-<tamanho_da_tela>-<orientacao>

// exemplos
res/layout-small-port
res/layout-normal-land
```

Como existe uma variedade grande de tamanhos de tela com

densidades distintas, o Android define uma unidade de medida virtual, o *density-independent pixel* (dp), que deve ser utilizada para expressar dimensões de layout e posições independentemente da densidade da tela. Um dp equivale a um pixel físico em uma tela de 160 dpi , que é a densidade padrão assumida pelo Android para uma tela de densidade média (mdpi).

Ele automaticamente faz as conversões e redimensionamentos das medidas em dp para pixels físicos, de acordo com a densidade da tela em uso.

Também existe uma medida relativa para a definição de tamanho dos textos, o sp (*scale-independent pixel*). Ele assemelha-se ao dp , com a diferença de que é redimensionado de acordo com as preferências de texto do usuário. O sp deve ser usado unicamente para a definição do tamanho de textos, nunca para dimensões de layout. Veja alguns exemplos de uso:

```
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/gastei"
    android:layout_marginTop="25dp" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/local"
    android:textSize="20sp" />
```

A partir da versão 3.2 do Android, a forma de definir o layout de acordo com o tamanho da tela (*small*, *normal*, *large* e *extra large*) foi depreciada. Agora podemos defini-los de acordo com a largura ou altura disponíveis em dp . É possível, por exemplo, definir um layout que deve ser exibido quando a largura da tela for de, no mínimo, 600 dp , colocando o arquivo XML correspondente no diretório res/layout-sw600dp .

O qualificador sw quer dizer *smallest width*. Sua largura

mínima refere-se à largura especificada pelo dispositivo (600x1024 `mdpi` em um tablet de 7", por exemplo). Portanto, seu valor não é alterado quando a orientação muda. Podemos utilizar o qualificador `w` para indicar a largura disponível, não importando se ela foi alterada por conta da mudança de orientação. Veja outros exemplos:

```
res/layout-w600dp/main.xml  // a largura deve ser de 600dp
res/layout-h320dp/main.xml  // altura mínima de 320dp
```

Se for necessário restringir para que o aplicativo não seja instalado em um dispositivo que não possua o tamanho de tela suportado, podemos incluir uma diretiva no `AndroidManifest.xml` que pode conter ainda a densidade de tela requerida. Veja como declarar essa restrição usando o `<supports-screens>`:

```
<supports-screens android:resizeable=["true"| "false"]
                  android:smallScreens=["true" | "false"]
                  android:normalScreens=["true" | "false"]
                  android:largeScreens=["true" | "false"]
                  android:xlargeScreens=["true" | "false"]
                  android:anyDensity=["true" | "false"]
                  android:requiresSmallestWidthDp="integer"
                  android:compatibleWidthLimitDp="integer"
                  android:largestWidthLimitDp="integer"/>
```

Nesse caso, você pode indicar para cada tipo de tela se ele será suportado ou não.

9.4 UTILIZE FRAGMENTS PARA SIMPLIFICAR SEUS LAYOUTS

Os `Fragments` são componentes modulares de interface gráfica com um ciclo de vida próprio, criados para facilitar o desenvolvimento de aplicativos com layouts ajustáveis a diferentes tamanhos de tela. Esse recurso foi adicionado no Android 3 e disponibilizado para as versões anteriores por meio de uma

biblioteca de compatibilidade.

A ideia é que um `Fragment` possa representar ora uma `Activity` única, sendo exibida em um dispositivo com tela compacta; ora uma parte de uma `Activity` que também exibe outros `Fragments`, no caso de um tablet. É o que demonstra a figura:

Figura 9.5: Exibição de Fragments

Um `Fragment` deve necessariamente fazer parte de uma `Activity` e o seu ciclo de vida é afetado por ela. Por exemplo, quando uma atividade é destruída, todos os seus `Fragments` também são.

Outra característica importante é que os `Fragments` podem ser adicionados e removidos da atividade em tempo de execução, trazendo flexibilidade para a criação de um fluxo de interação com o usuário, que leva em conta o dispositivo que está sendo utilizado. Para adicionar um `Fragment` a uma `Activity`, podemos declará-lo no layout dela, por meio do elemento `<fragment>`, ou adicioná-lo programaticamente a um `ViewGroup` (todos os layouts herdam desta classe) já existente.

Retornaremos ao aplicativo BoaViagem para implementar uma nova funcionalidade que tirará proveito da tela maior de um tablet. Como uma espécie de diário de bordo, essa funcionalidade consiste

em registrar anotações diversas sobre a viagem. A figura a seguir ilustra como os `fragments` serão usados:

Figura 9.6: Utilizando Fragments

Como o foco é o uso da API de `Fragments`, não vamos nos preocupar em recuperar as informações do banco de dados e nem criar layouts personalizados para as listagens. Começaremos esta nova versão implementando um `Fragment` para a lista de viagens. Por questões de organização, crie um novo pacote `br.com.casadocodigo.boaviagem.fragment` para manter todos os `Fragments` que serão criados.

De forma análoga à `ListActivity`, para utilizar uma `ListView`, podemos estender a classe `ListFragment`. Crie, então, uma nova classe com nome de `ViagemListFragment` que estende de `ListFragment`. Além disso, para tratar o evento de quando um item da lista for selecionado, nossa nova classe deverá implementar `OnItemClickListener`:

```
public class ViagemListFragment extends ListFragment
                                implements OnItemClickListener {
    @Override
```

```
    public void onStart() {
    }

    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
    }
}
```

Diferentemente do método `onCreate` da `Activity`, o método `onCreate` de um `Fragment` é executado antes da criação da sua `view` correspondente; o que em outras palavras quer dizer que não temos como acessar componentes de interface gráfica nesse método, pois eles ainda nem foram criados. Por isso, vamos sobrescrever o método `onStart`, que é executado quando o `Fragment` já está pronto para ser exibido, para realizar as implementações necessárias. Veja o código:

```
@Override
public void onStart() {
    super.onStart();
    List<String> viagens = Arrays.asList("Campo Grande",
      "São Paulo", "Miami");
    ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(getActivity(),
        android.R.layout.simple_list_item_1, viagens);
    setListAdapter(adapter);
    getListView().setOnItemClickListener(this);
}
```

No método `onStart`, criamos uma lista simulada de viagens e um `ArrayAdapter` com a lista criada para alimentar a `ListView` que será exibida. Também atribuímos o próprio `Fragment` como responsável por tratar os eventos de seleção de itens da listagem.

O mínimo necessário para o nosso primeiro `Fragment` já foi implementado. Na sequência, vamos criar uma `Activity` para controlar a exibição dos `Fragments` que compõem essa nova funcionalidade. Para isso, precisaremos criar dois layouts: um deles será usado para tablets e outro para dispositivos com telas menores.

No layout destinado a tablets, utilizaremos três `Fragments` ; enquanto no outro, apenas um. Crie um novo XML de layout, chamado `anotacoes.xml` , e coloque-o no diretório `res/layout` . Sua definição é a seguinte:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <FrameLayout
        android:id="@+id/fragment_unico"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />

</LinearLayout>
```

Repare que não declaramos um elemento `<fragment>` no layout. Em vez disso, incluímos um `FrameLayout` , o qual será substituído em tempo de execução pelo `Fragment` que deve ser exibido.

Para o tablet, crie um novo layout com o mesmo nome (`anotacoes.xml`), porém, coloque-o no diretório `res/layout-large-land` . Esse layout será utilizado quando o dispositivo estiver com a orientação paisagem e possuir uma tela classificada como grande.

Nele, definiremos três `fragments` , mas como ainda não criamos todos eles, declararemos apenas um, e dois `FrameLayouts` para simular o espaço ocupado pelos outros `fragments` :

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/fragment_viagens"
        android:name=
```

```
"br.com.casadocodigo.boaviagem.fragment.ViagemListFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/fragment_anotacoes"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="FRAGMENT B" />
    </FrameLayout>

    <FrameLayout
        android:id="@+id/fragment_anotacao"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="FRAGMENT C" />
    </FrameLayout>

</LinearLayout>
```

O atributo `name` do elemento `<fragment>` deve ser o nome qualificado da classe que o implementa. O próximo passo será criar a classe `AnotacaoActivity`, que controlará a exibição dos `fragments` e a comunicação entre eles. Iniciaremos a implementação dessa classe com o código a seguir:

```
public class AnotacaoActivity extends FragmentActivity {

    private boolean tablet = true;

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.anotacoes);
```

```
        View view = findViewById(R.id.fragment_unico);

        if(view != null){
            tablet = false;

            ViagemListFragment fragment =
        new ViagemListFragment();
            fragment.setArguments(bundle);

            FragmentManager manager =
          getSupportFragmentManager();
            FragmentTransaction transaction =
                    manager.beginTransaction();
            transaction.replace(R.id.fragment_unico, fragment);
            transaction.addToBackStack(null);
            transaction.commit();
        }
    }
}
```

Repare que a classe estende de `FragmentActivity`, em vez de `Activity`. A classe `FragmentActivity` pertence ao pacote de compatibilidade e, por meio dela, podemos usar os recursos da API de `Fragments` em versões do Android anteriores à versão 3. No entanto, se o seu aplicativo tiver como alvo esta ou versão superior, então você deve utilizar a classe `Activity` que já implementa a API nessas versões.

No método `onCreate`, atribuímos a `view` identificada por `R.layout.anotacaoes` para a `activity`. Em tempo de execução, o Android decidirá se vai utilizar o `anotacoes.xml` da pasta `layout` ou aquele localizado em `layout-large-land`, de acordo com o dispositivo que está em uso. Por isso, tentamos recuperar o `FrameLayout` com o id `R.id.fragment_unico` para determinar qual layout está sendo usado. Se esta `view` existir, o dispositivo em uso não é um tablet e é necessário exibir um `Fragment` por vez.

Em seguida, instanciamos o `ViagemListFragment`, que é o primeiro a ser exibido para o usuário. Na linha seguinte, atribuímos

um `bundle` que é utilizado para a construção e inicialização do `Fragment`. O que é feito em seguida é substituir o `FrameLayout` existente pelo recém-criado `ViagemListFragment`. Para isso, usamos uma `FragmentTransaction` que é iniciada através do `FragmentManager`.

Por fim, invocamos o método `replace` da `FragmentTransaction` informando no primeiro parâmetro qual é a `view` que deve ser substituída e, no segundo, qual é o `Fragment` que deve ser colocado em seu lugar. O método `addToBackStack` é utilizado para incluir a transação na *back stack*; o que, na prática, quer dizer que este `Fragment` estará disponível quando o usuário estiver em outra tela e pressionar o botão "Voltar". Logo, é feito o `commit` da transação para efetivar a operação de substituição.

Quando a aplicação estiver executando em um tablet, nada de diferente precisa ser feito, pois no layout já temos a declaração dos `Fragments` que devem ser exibidos. Para executar esta nova funcionalidade, declare a `AnotacaoActivity` no manifesto e crie uma nova opção na dashboard para iniciá-la. Usando o emulador de tablets criado previamente para executar o aplicativo, o resultado obtido deve ser semelhante ao exibido na figura:

Figura 9.7: Fragment da lista de viagens

Dando continuidade, vamos implementar os demais `Fragments` e programar a comunicação entre eles. Quando o usuário selecionar uma viagem da lista, devemos exibir as anotações associadas àquela viagem no segundo `Fragment` . Ao selecionar uma anotação da lista, o terceiro `Fragment` exibirá as informações daquela anotação. Esse é o funcionamento desejado para o tablet. Caso contrário, cada `Fragment` deverá ser exibido individualmente e quem controlará isto será a `AnotacaoActivity` .

O segundo `Fragment` , além de exibir uma `ListView` com as anotações já realizadas, também terá uma botão que permite a criação de uma nova anotação. Então, precisaremos de um layout que possua um botão e uma `ListView` . Ele será usado pelo `Fragment` de anotações tanto para smartphones como para tablets.

Crie um novo XML no diretório `res/layout` com nome de `lista_anotacoes.xml` , dessa forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/nova_anotacao"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/nova_anotacao" />

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </ListView>

</LinearLayout>
```

Para utilizarmos um `ListFragment` com um layout que contém mais widgets, além da `ListView`, é obrigatório declarar uma `ListView` com o `id` `@android:id/list`.

Com o layout já definido, crie uma nova classe para a lista de anotações com o nome de `AnotacaoListFragment`. O código inicial dessa classe será o seguinte:

```
public class AnotacaoListFragment extends ListFragment
        implements OnItemClickListener, OnClickListener {

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.lista_anotacoes,
                                container, false);
    }

    @Override
    public void onStart() {
        super.onStart();
    }

    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
```

```
    }

    @Override
    public void onClick(View v) {
    }

}
```

O layout que será usado pelo `Fragment` é construído no `onCreateView`, que cria a `view` por meio do método `LayoutInflater.inflate`.

A `AnotacaoListFragment` implementa duas interfaces: a `OnItemClickListener`, para tratar eventos da lista de anotações; e a `OnClickListener`, para tratar o evento do botão que cria uma nova anotação. Quando estávamos lidando apenas com *activities*, nós colocávamos no atributo `onClick` do elemento `<Button>` o nome do método que deveria ser invocado. No entanto, este recurso não está disponível para os `Fragments`.

No método `onStart`, além de criar uma lista e um *adapter* para a `ListView`, também precisaremos atribuir o `onClickListener`, que é o próprio `Fragment`, para o botão `R.id.nova_anotacao`. Veja como fica a implementação desse método:

```
@Override
public void onStart() {
    super.onStart();
    List<Anotacao> anotacoes = listarAnotacoes();

    ArrayAdapter<Anotacao> adapter =
            new ArrayAdapter<Anotacao>(getActivity(),
                    android.R.layout.simple_list_item_1,
                    anotacoes);

    setListAdapter(adapter);
    getListView().setOnItemClickListener(this);

    Button button =
        (Button) getActivity().findViewById(R.id.nova_anotacao);
    button.setOnClickListener(this);
```

```java
}

private List<Anotacao> listarAnotacoes() {

    List<Anotacao> anotacoes = new ArrayList<Anotacao>();

    for (int i = 1; i <= 20; i++) {
        Anotacao anotacao = new Anotacao();
        anotacao.setDia(i);
        anotacao.setTitulo("Anotacao " + i);
        anotacao.setDescricao("Descrição " + i);
        anotacoes.add(anotacao);
    }

    return anotacoes;
}
```

O `ArrayAdapter` invoca o método `toString` dos objetos da lista passada como parâmetro, para obter o valor que deve ser colocado em cada linha da `ListView` . Assim, sobrescrevemos o método `toString` da `Anotacao` para retornar seu dia e título.

A classe `Anotacao` , que representa uma anotação feita pela usuário, é definida como:

```java
public class Anotacao{

    private Long id;
    private Integer dia;
    private String titulo;
    private String descricao;

    // getters e setters

    @Override
    public String toString() {
        return "Dia " + dia + " - " + titulo;
    }
}
```

Para que o novo `AnotacaoListFragment` seja visualizado, precisamos incluí-lo no layout para tablets. Basta alterar o arquivo `anotacoes.xml` do diretório `layout-large-land` , para indicar o novo `fragment` criado:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <!-- fragment ViagemListFragment -->

    <fragment
        android:id="@+id/fragment_anotacoes"
        android:name=
    "br.com.casadocodigo.boaviagem.fragment.AnotacaoListFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <!-- FrameLayout FRAGMENT C -->

</LinearLayout>
```

Ao executar a aplicação no emulador de tablet, veremos uma tela semelhante à apresentada na figura a seguir. No entanto, se o tablet estiver na orientação retrato, ou um smartphone estiver executando o aplicativo, o único `Fragment` exibido continua sendo o da lista de viagens. Antes de nos preocuparmos com essa questão, vamos primeiro finalizar o último `Fragment`, que exibirá os detalhes da anotação selecionada.

Figura 9.8: Fragment da lista de viagens

Mudando a orientação no emulador

No emulador, para mudar a orientação entre retrato e paisagem (e vice-versa), basta pressionar `Ctrl+F12` .

Crie um novo XML de layout com nome de `anotacao.xml` , no local padrão para ser utilizado pelo `Fragment` de detalhes da anotação. Ele terá widgets básicos, como `TextViews` , para a descrição dos campos, `EditTexts` para o usuário informar os dados, além de um `Button` , para confirmar.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/dia" />

    <EditText
        android:id="@+id/dia"
        android:layout_width="50dp"
        android:layout_height="wrap_content"
        android:inputType="number" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/titulo" />

    <EditText
        android:id="@+id/titulo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/descricao" />

    <EditText
        android:id="@+id/descricao"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/salvar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/salvar"/>

</LinearLayout>
```

Como, neste caso, não precisaremos de uma `ListView`, vamos criar uma nova classe, que herda de `Fragment`, com o nome de `AnotacaoFragment`. Também implementaremos `OnClickListener` para tratar a operação de salvar. O código inicial dessa classe será o seguinte:

```java
public class AnotacaoFragment extends Fragment
                implements OnClickListener {

    private EditText dia, titulo, descricao;
    private Button botaoSalvar;
    private Anotacao anotacao;

    @Override
    public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.anotacao,
                    container, false);
    }

    @Override
    public void onStart() {
        super.onStart();

        dia = (EditText) getActivity().findViewById(R.id.dia);
        titulo =
      (EditText) getActivity().findViewById(R.id.titulo);
        descricao =
    (EditText) getActivity().findViewById(R.id.descricao);
        botaoSalvar =
  (Button) getActivity().findViewById(R.id.salvar);
        botaoSalvar.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        // salvar Anotacao no banco de dados
    }
}
```

Agora basta incluir o `AnotacaoFragment` no `anotacoes.xml` , que ficará assim:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/fragment_viagens"
        android:name=
    "br.com.casadocodigo.boaviagem.fragment.ViagemListFragment"
        android:layout_width="0dp"
```

```
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/fragment_anotacoes"
        android:name=
    "br.com.casadocodigo.boaviagem.fragment.AnotacaoListFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/fragment_anotacao"
        android:name=
        "br.com.casadocodigo.boaviagem.fragment.AnotacaoFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

</LinearLayout>
```

Ao executar a aplicação novamente, já visualizaremos o layout completo com todos os `Fragments` criados, como mostra a figura:

Figura 9.9: Fragment da lista de viagens

9.5 COMUNICAÇÃO ENTRE FRAGMENTS

Já construímos os `Fragments` necessários para o registro de anotações. Nesta seção, veremos como trocar informações entre `Fragments` para completar a implementação desta nova funcionalidade do aplicativo BoaViagem.

Para que possamos facilmente reutilizar um `Fragment`, é necessário que ele seja implementado como um componente modular, possuindo o seu próprio layout e comportamento. Dessa forma, podemos usar o mesmo `Fragment` para compor telas distintas que variam conforme o dispositivo em uso.

Um `Fragment` sempre está atrelado a uma `Activity`, e é ela quem deve orquestrar a comunicação entre seus `Fragments`, inclusive controlar quais deles devem ser exibidos. Já iniciamos uma implementação assim na `AnotacaoActivity`, que verifica se serão exibidos vários `Fragments` ou apenas um.

Para completar a funcionalidade de registro de anotações, nesta `Activity` implementaremos a lógica de trocar o `Fragment` exibido, quando o aplicativo estiver sendo utilizado em um smartphone e for necessário apresentar um por vez. Também faremos a comunicação e a passagem de dados de um para outro, usando essa `Activity`.

Os `Fragments` não devem se comunicar diretamente; por isso, é recomendado que seja definida uma `interface` com as operações que eles devem notificar. A `Activity` responsável por coordená-los deve implementar essa `interface` e tomar as ações necessárias, geralmente invocando algum método de outro `Fragment`.

No nosso caso, o `ViagemListFragment` deve notificar a `AnotacaoActivity`, quando uma viagem for selecionada na lista. O mesmo acontece com a `AnotacaoListFragment`, que além de informar qual item da `ListView` foi selecionado, também notifica

quando o usuário pressiona o botão para criar uma nova anotação. Vamos criar uma `interface` com nome de `AnotacaoListener` para definir esses comportamentos. Veja o código:

```
public interface AnotacaoListener {
    void viagemSelecionada(Bundle bundle);
    void anotacaoSelecionada(Anotacao anotacao);
    void novaAnotacao();
}
```

No `ViagemListFragment`, invocaremos o método `viagemSeleciona` quando o usuário escolher um item da lista. Para isso, precisaremos de uma referência para a `Activity` que implementa a `AnotacaoListener`. Podemos obtê-la sobrescrevendo o método `onAttach` do `Fragment`. Na implementação do método `onItemClick`, recuperaremos o item selecionado da `ListView`, colocando-o em um `Bundle`, que será passado como parâmetro do método `viagemSelecionada`. Veja como ficou:

```
public class ViagemListFragment extends ListFragment
            implements OnItemClickListener {
    // novo atributo
    private AnotacaoListener callback;

    // códigos existentes

    @Override
    public void onItemClick(AdapterView<?> parent,
                            View view, int position,
                            long id) {
        String viagem =
      (String) getListAdapter().getItem(position);
        Bundle bundle = new Bundle();
        bundle.putString(Constantes.VIAGEM_SELECIONADA, viagem);
        callback.viagemSelecionada(bundle);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        callback = (AnotacaoListener) activity;
    }
```

```
}

public class Constantes {
    // nova constante
    public static final String VIAGEM_SELECIONADA =
                "viagem_selecionada";
}
```

Na classe AnotacaoListFragment , precisaremos disponibilizar um método, que será chamado pela AnotacaoActivity , para listar as anotações de uma viagem, recebendo por parâmetro o bundle que foi passado pelo outro Fragment contendo a viagem selecionada.

Criaremos um método chamado listarAnotacoesPorViagem , que verificará se alguma viagem foi informada por meio do bundle recebido como parâmetro. Ele cria uma lista de anotações fictícias para alimentar a ListView ; no entanto, seria facilmente adaptável para recuperar as informações do banco de dados, caso necessário.

Modificaremos o método onStart , para também utilizar o listarAnotacoesPorViagem . Dessa forma, quando o Fragment for inicializado programaticamente pela AnotacaoActivity , a lista de anotações também será carregada.

```
@Override
public void onStart() {
    super.onStart();
    Button button =
        (Button) getActivity().findViewById(R.id.nova_anotacao);
    button.setOnClickListener(this);
    getListView().setOnItemClickListener(this);
    listarAnotacoesPorViagem(getArguments());
}

public void listarAnotacoesPorViagem(Bundle bundle) {
    if(bundle != null &&
        bundle.containsKey(Constantes.VIAGEM_SELECIONADA)){

        //utilize a informação do bundle para buscar
        // as anotacoes no banco de dados
```

```
        List<Anotacao> anotacoes = listarAnotacoes();

        ArrayAdapter<Anotacao> adapter =
                new ArrayAdapter<Anotacao>(getActivity(),
                        android.R.layout.simple_list_item_1,
                        anotacoes);

        setListAdapter(adapter);
    }
}
```

O próximo passo é alterar a `AnotacaoActivity` para implementar `AnotacaoListener`, deste modo:

```
public class AnotacaoActivity extends FragmentActivity
                              implements AnotacaoListener {

    // códigos existentes

    @Override
    public void viagemSelecionada(Bundle bundle) { }

    @Override
    public void anotacaoSelecionada(Anotacao anotacao) {}

    @Override
    public void novaAnotacao() {}
}
```

No método `viagemSelecionada`, vamos verificar se o aplicativo está sendo executado em um tablet ou em outro dispositivo com uma tela maior. Assim, com base nessa informação, vamos recuperar o `AnotacaoListFragment` e invocar o seu método `listarAnotacoesPorViage` (caso seja tablet); ou vamos instanciar um novo `AnotacaoListFragment` e colocá-lo no lugar do `FrameLayout` (caso seja um dispositivo com tela menor).

Obteremos um `FragmentManager` que será usado para realizar operações sobre os `Fragments`, por meio do método `getSupportFragmentManager`.

Caso um tablet esteja sendo utilizado, recuperamos o `Fragment` com a lista de anotações através do `manager` e, na linha

seguinte, invocamos o método para listar anotações de uma determinada viagem, para assim atualizar a lista de anotações exibidas pelo `AnotacaoListFragment` .

Caso não seja um tablet, criamos um novo `AnotacaoListFragment` , atribuímos a ele um `bundle` e, por meio de uma `transaction` , substituímos o `FrameLayout` pelo `Fragment` recém-criado. Dessa forma, quando o usuário selecionar uma viagem, haverá uma transição do `ViagemListFragment` para esse novo `Fragment` , que será exibido individualmente na tela.

```
@Override
public void viagemSelecionada(Bundle bundle) {

    FragmentManager manager =
      getSupportFragmentManager();
    AnotacaoListFragment fragment;

    if(tablet) {
        fragment = (AnotacaoListFragment) manager
                .findFragmentById(R.id.fragment_anotacoes);
        fragment.listarAnotacoesPorViagem(bundle);

    } else {
        fragment = new AnotacaoListFragment();
        fragment.setArguments(bundle);

        manager.beginTransaction()
                    .replace(R.id.fragment_unico, fragment)
                    .addToBackStack(null)
                    .commit();
    }
}
```

Os outros dois métodos da `AnotacaoListener` serão implementados de forma bastante similar ao que acabamos de fazer. A próxima implementação será para tratar a seleção de uma anotação, cujos detalhes devem ser exibidos no `AnotacaoFragment` . Na classe `AnotacaoListFragment` , implementaremos o método `onItemClick` para recuperar um objeto `Anotacao` da `ListView` e passá-lo como parâmetro para o

método anotacaoSelecionada .

```
// novo atributo
private AnotacaoListener callback;

// códigos existentes

@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    callback = (AnotacaoListener) activity;
}

@Override
public void onItemClick(AdapterView<?> parent,
                        View view, int position,
                        long id) {

    Anotacao anotacao =
    (Anotacao) getListAdapter().getItem(position);
    callback.anotacaoSelecionada(anotacao);
}
```

Em vez de utilizar um Bundle , dessa vez passamos o próprio objeto recuperado da lista como parâmetro para o método anotacaoSelecionada . É comum o uso das duas alternativas; cabe a você decidir, de acordo com a sua aplicação, qual a melhor estratégia a ser adotada.

No AnotacaoFragment , criamos dois métodos públicos: um para atribuir a anotação, que deve ser exibida pelo Fragment ; e outro para preparar a edição de uma anotação selecionada da lista. Veja o código alterado da classe AnotacaoFragment :

```
// Novo atributo
private Anotacao anotacao;

@Override
public void onStart() {
    super.onStart();

    dia = (EditText) getActivity().findViewById(R.id.dia);
    titulo =
    (EditText) getActivity().findViewById(R.id.titulo);
```

```
    descricao =
    (EditText) getActivity().findViewById(R.id.descricao);
    botaoSalvar =
    (Button) getActivity().findViewById(R.id.salvar);
    botaoSalvar.setOnClickListener(this);

    if(anotacao != null) {
        prepararEdicao(anotacao);
    }
}

public void setAnotacao(Anotacao anotacao) {
    this.anotacao = anotacao;
}

public void prepararEdicao(Anotacao anotacao) {
    setAnotacao(anotacao);
    dia.setText(anotacao.getDia().toString());
    titulo.setText(anotacao.getTitulo());
    descricao.setText(anotacao.getDescricao());
}
```

Retornando para a `AnotacaoActivity` , implementaremos o método `anotacaoSelecionada` da seguinte forma:

```
@Override
public void anotacaoSelecionada(Anotacao anotacao) {
    FragmentManager manager =
      getSupportFragmentManager();
    AnotacaoFragment fragment;

    if(tablet){
        fragment = (AnotacaoFragment) manager
            .findFragmentById(R.id.fragment_anotacao);
        fragment.prepararEdicao(anotacao);

    }else{
        fragment = new AnotacaoFragment();
        fragment.setAnotacao(anotacao);

        manager.beginTransaction()
                    .replace(R.id.fragment_unico, fragment)
                    .addToBackStack(null)
                    .commit();
    }
}
```

A implementação é bem parecida com a realizada no método `viagemSelecionada`, com a diferença apenas do `Fragment` usado e dos métodos invocados. Para implementar a regras associadas à criação de uma nova anotação, também será bastante simples. Na `AnotacaoListFragment` precisamos implementar o método `onClick` que fará o `callback` sem informar nenhum parâmetro:

```
@Override
public void onClick(View v) {
    callback.novaAnotacao();
}
```

Criaremos um método público em `AnotacaoFragment` para ser invocado pela `AnotacaoActivity`, quando o usuário escolher a opção de criar uma nova anotação. Sua função é basicamente limpar quaisquer dados que estavam sendo exibidos previamente. Veja a sua implementação:

```
public void criarNovaAnotacao() {
    anotacao = new Anotacao();
    dia.setText("");
    titulo.setText("");
    descricao.setText("");
}
```

Na `AnotacaoActivity`, invocaremos o método `criarNovaAnotacao`, caso seja um tablet que já está exibindo todos os `Fragments`, ou criaremos um novo `AnotacaoFragment` sem informar uma anotação, já que desejamos criar uma nova. Veja como fica a implementação do método `novaAnotacao`:

```
@Override
public void novaAnotacao() {
    FragmentManager manager =
    getSupportFragmentManager();
    AnotacaoFragment fragment;

    if(tablet) {
        fragment = (AnotacaoFragment) manager
            .findFragmentById(R.id.fragment_anotacao);
        fragment.criarNovaAnotacao();
```

```
    } else {
        fragment = new AnotacaoFragment();
        manager.beginTransaction()
                .replace(R.id.fragment_unico, fragment)
                .addToBackStack(null)
                .commit();
    }
}
```

Pronto! Nossos `Fragments` já estão implementados e interagem entre si. Execute a aplicação e confira o resultado!

9.6 CARREGUE DADOS COM LOADERS

Outro recurso muito útil inserido na versão 3 do Android e disponível no pacote de compatibilidade são os `Loaders`, que possuem a função de carregar dados de forma assíncrona em uma `Activity` ou `Fragment`. Outra característica é que ele pode monitorar uma fonte de dados, como um `ContentProvider`, e automaticamente disponibilizar os resultados, assim que a informação é atualizada.

O Android já disponibiliza o `CursorLoader` para carregar dados em `cursores` e, se necessário, podemos implementar o nosso próprio `Loader`, estendendo de `AsyncTaskLoader`, para carregar assincronamente outros tipos de recurso.

Nesta seção, faremos uma implementação de exemplo para explorar esse recurso, utilizando o `ContentProvider` que já criamos para carregar a lista de viagens por meio de um `CursorLoader`. A classe `LoaderManager` é a responsável por gerenciar um ou mais `Loaders` em uma `Activity` ou `Fragment`. As operações devem ser realizadas através de `callbacks`. Para isso, é necessário implementar a interface `LoaderCallbacks`. Veja o código abaixo:

```
public class ViagemListFragment extends ListFragment
            implements    OnItemClickListener,
```

```
        LoaderCallbacks<Cursor> {
    // códigos existentes

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        getLoaderManager().initLoader(0, null, this);
    }

    Override
    public Loader<Cursor> onCreateLoader(
      int id, Bundle args) { }

    @Override
    public void onLoadFinished(
      Loader<Cursor> loader, Cursor data) { }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) { }
}
```

Para implementar `LoaderCallbacks`, incluímos o método `onCreateLoader`, que é usado para criar um novo `Loader`, o `onLoadFinished` e o `onLoaderReset`, que são chamados, respectivamente, quando um `Loader` terminou de ser carregado e foi resetado.

No método `onActivityCreated`, invocado quando a atividade que contém o `Fragment` é criada, recuperamos um `LoaderManager` e fazemos a inicialização de um `Loader` por meio do método `initLoader`. O primeiro parâmetro que esse método recebe é um número que identifica o `Loader` que se deseja carregar. No segundo, podemos passar um `Bundle` com dados necessários para sua construção e, o terceiro argumento, é a implementação do `callback` que, neste caso, é o próprio `Fragment`.

No método `onCreateLoader`, devemos instanciar e retornar o `Loader` que será usado. No nosso caso, utilizaremos um `CursorLoader` para carregar assincronamente as viagens obtidas

do nosso ContentProvider . A implementação desse método é a seguinte:

```
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    Uri uri = Viagem.CONTENT_URI;
    String[] projection =
      new String[]{ Viagem._ID, Viagem.DESTINO };
    return new CursorLoader(getActivity(),
            uri, projection, null, null, null);
}
```

Por enquanto, só estamos carregando os dados e ainda não os colocamos na ListView . O próximo passo é alterar a implementação do método onStart , para utilizar um CursorAdapter :

```
// novo atributo
private SimpleCursorAdapter adapter;

@Override
public void onStart() {
    super.onStart();
    adapter = new SimpleCursorAdapter(getActivity(),
            android.R.layout.simple_list_item_1, null,
            new String[] { Viagem.DESTINO },
            new int[] { android.R.id.text1 }, 0);
    setListAdapter(adapter);
    getListView().setOnItemClickListener(this);
}
```

Quando o carregamento dos dados for finalizado, no método onLoaderFinished devemos alterar o cursor usado pelo CursorAdapter , para exibir os dados recuperados. Da mesma forma, devemos alterá-lo quando o Loader for reiniciado. Veja como ficam essas implementações:

```
@Override
public void onLoadFinished(
    Loader<Cursor> loader, Cursor data) {
    adapter.swapCursor(data);
}

@Override
```

```
public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
}
```

A última alteração necessária é modificar a implementação do `onItemClick` para recuperar o `id` do item selecionado na lista de viagens. Esse `id` será utilizado posteriormente para carregar as anotações associadas com a viagem selecionada. As alterações são as seguintes:

```
@Override
public void onItemClick(AdapterView<?> parent,
     View view, int position,
     long id) {
  long viagem = getListAdapter().getItemId(position);
  Bundle bundle = new Bundle();
  bundle.putLong(Constantes.VIAGEM_SELECIONADA, viagem);
  callback.viagemSelecionada(bundle);
}
```

Agora, ao executar a aplicação, teremos o `Loader` carregando os dados para alimentar a lista de viagens. Experimente!

9.7 CONCLUSÃO

Neste capítulo, vimos como projetar nosso aplicativo para que ele seja compatível com tablets e outros dispositivos, com diversos tamanhos de tela e diferentes versões de Android.

Acrescentamos mais uma funcionalidade ao aplicativo BoaViagem, que usa os recursos da API de `Fragments` para prover uma interação, e layouts diferenciados para o uso em um tablet. Por fim, exercitamos a utilização de `Loaders` para carregar dados de forma assíncrona.

Capítulo 10

DESENVOLVA PARA TODAS AS VERSÕES DO ANDROID

O sistema operacional Android, desde o seu lançamento, tem mostrado um enorme crescimento da base instalada; ou seja, houve aumento da quantidade de dispositivos rodando alguma de suas versões. Além disso, o sistema passou por várias atualizações com incrementos importantes na plataforma tanto para os usuários finais quanto para os desenvolvedores de aplicativos e jogos. Desde sua criação, passando pela versão 1.5 (Cupcake) até a mais atual 5.1 (Lollipop), já se foram dezenas de versões.

No entanto, esse crescimento aliado às constantes (e boas) atualizações levou ao fenômeno conhecido como fragmentação, no qual temos um grande número de dispositivos rodando as mais diversas versões do Android. Na prática, o que acontece é que nós, desenvolvedores, precisamos nos preocupar em suportar o maior número de versões possíveis e lidar com características de cada uma, bem como contornar limitações da API nas mais antigas. Neste capítulo, veremos como compatibilizar nosso aplicativo para diversas versões do Android.

10.1 UTILIZE A ANDROID SUPPORT LIBRARY

Até pouco tempo atrás, a fragmentação era uma das grandes preocupações do desenvolvedor Android. Isso fez com que a plataforma recebesse duras críticas. Para sanar esse problema, o Google lançou mão da chamada *Android Support Library*. A ideia é empacotar os novos recursos do Android não disponíveis em versões anteriores em bibliotecas que o desenvolvedor possa usar na sua aplicação. Dessa forma, é possível, por exemplo, utilizar uma `ActionBar` no Android 2.3 através da biblioteca de suporte; enquanto antes só era possível fazê-lo a partir da versão 3.0 (Honeycomb) do Android, ou usando bibliotecas de terceiros (como a *ActionBarSherlock*).

Cada biblioteca de suporte é retrocompatível com uma versão específica da API Android. Por isso, é importante definir quais recursos são desejados no seu aplicativo e quais versões serão suportadas, para fazer a escolha mais adequada. Usar as bibliotecas de suporte nos auxilia a enriquecer o visual do nosso aplicativo por meio de recursos novos de UI, melhorar a performance e oferecer isso tudo para boa parte da base instalada de dispositivos. Enquanto este livro estava sendo escrito, existiam 5 bibliotecas de suporte voltadas para versões diferentes do Android, como veremos a seguir.

- **v4 Support Library —** a biblioteca de suporte v4 é compatível com dispositivos rodando Android 1.6 (API 4, daí o nome v4) e é a biblioteca com o maior conjunto de funcionalidades, disponibilizando recursos de acessibilidade, interface gráfica, componentes de aplicação e manipulação de conteúdo. Dois recursos destacam-se nela: os `Fragments` e os `Loaders` .

- **v7 Support Library —** projetada para suportar o Android 2.1 e superiores, esta biblioteca é composta pelos módulos: `v7-appcompat` , `v7-gridlayout` ,

`v7-mediarouter` , `v7-cardview` , `v7-recyclerview` e `v7-pallete` . A `v7-appcompat` disponibiliza a `ActionBar` , cujo uso se tornou padrão em qualquer aplicativo Android. A `v7 gridlayout` possibilita a criação de interfaces gráficas, agrupando elementos em células retangulares através do `GridLayout` . A `v7-mediarouter` possui os recursos necessário para manipular *streams* e canais de mídia, como por exemplo, controlar o que deve ser exibido e tocado em telas e alto-falantes conectados ao dispositivo. A `v7-cardview` e a `v7-recyclerview` são novos widgets adicionados no Android 5.0 e que serão detalhados nos próximos capítulos. E, por fim, a `v7-palette` permite obter cores a partir de imagens.

- **v8 Support Library —** esta biblioteca permite a utilização do RenderScript a partir da API 8. O RenderScript é um framework para processamento de tarefas intensivas do ponto de vista computacional. Por meio do RenderScript, é possível paralelizar as tarefas e usar todos os processadores disponíveis no dispositivo, incluindo a CPU e a GPU. O RenderScript é especialmente útil para aplicações que necessitam intensamente de processamento de imagens.
- **v13 Support Library —** suporta Android 3.2 e adiciona classes utilitárias para lidar com `Fragments` .
- **v17 Support Library —** provê APIs para construir interfaces gráficas voltadas para TV.

De forma geral, para tornar nosso aplicativo compatível com o Android 2.1 e superiores, é necessário usar apenas a bibliotecas `v4` e `v7-appcompat` . Elas estão disponíveis para serem baixadas e

atualizadas por meio do SDK Manager e encontram-se localizadas na seção Extras, apresentadas como sendo apenas um único item. Ao realizar o download desse item, as bibliotecas de compatibilidade estarão localizadas no diretório <sdk>/extras/android/support/ . Veja na figura a seguir a janela do SDK Manager com a biblioteca de suporte selecionada. Na próxima seção, veremos como adicionar a biblioteca em nosso projeto.

Figura 10.1: SDK Manager

10.2 INCORPORE A ACTIONBAR EM SEU APLICATIVO

A ActionBar , ou barra de ação, tornou-se um dos principais elementos de interface gráfica e hoje está presente na grande maioria dos aplicativos. Sua principal função é identificar em qual região do aplicativo o usuário encontra-se, além de fornecer ações

contextualizadas que o usuário pode executar, bem como um esquema de navegação eficiente para outras regiões do aplicativo. Além disso, a `ActionBar` também é capaz de ajustar-se adequadamente a qualquer configuração de tela. A figura a seguir ilustra os elementos existentes em uma `ActionBar` .

Figura 10.2: Elementos da ActionBar. Fonte: http://developer.android.com

1. **Ícone —** O ícone é o elemento que determina a identidade do seu aplicativo. Aqui você pode utilizar o mesmo ícone exibido no *launcher*, ou, se preferir, um logotipo diferente.

2. **Navegação —** Esta parte da `ActionBar` permite que o usuário selecione outras telas (*views*) do aplicativo, permitindo, assim, uma rápida navegação. Para implementar a troca de views, é possível utilizar menus do tipo *drop-down*, como mostrado na figura, ou abas.

3. **Botões de ação —** Exibe as opções (ações) mais importantes disponíveis para o usuário. As ações que não couberem na barra são movidas automaticamente para uma outra região, chamada de *action overflow*.

4. **Action overflow —** É uma espécie de submenu que reúne as ações menos usadas ou que não couberam na barra de ações.

Uma das características da `ActionBar` é permitir que o seu conteúdo seja divido entre regiões da tela. Essa funcionalidade é particularmente útil para lidar de forma adequada com a rotação do dispositivo e com os diferentes tamanhos de tela. É possível colocar três barras de ação em uma única tela, como por exemplo: incluir uma barra de ação principal, na parte superior da tela; outra

secundária, logo abaixo da principal; e, por último, uma terceira barra de ação, localizada na parte inferior da tela.

Os botões de ação da `ActionBar` devem listar as funcionalidades mais importantes do seu aplicativo; por isso, é essencial que eles estejam ordenados de acordo com a sua relevância e frequência de uso. Ou seja, as ações mais usadas e importantes devem vir primeiro; e as menos importantes e utilizadas, ir para o *action overflow*. Outra recomendação importante sugerida pelo Google é mostrar apenas ações que podem ser realizadas naquele momento e, na `ActionBar` , jamais devem ser exibidas opções como desabilitadas.

Outra dica é utilizar ícones pré-definidos para ações que são usuais em qualquer aplicativo, como o compartilhar, atualizar e download. Isso auxilia o usuário a rapidamente identificar o que a ação significa. Para facilitar a criação de outros ícones para a barra de ação, o Google disponibiliza um pacote com imagens que podem ser customizadas em http://developer.android.com/design/downloads/index.html.

Ao se criar um novo projeto usando o Android Studio, existe a opção de se criar uma `Activity` por meio de um template que já traz uma `ActionBar` . Dessa forma, a IDE automaticamente incluirá as bibliotecas de suporte necessárias para a utilização da `ActionBar` . Confira na figura a seguir qual template deve ser selecionado.

Figura 10.3: Criando nova Activity com ActionBar

Outra alternativa é adicionar as bibliotecas de suporte manualmente. Isso, provavelmente, será necessário se você já possui uma aplicação e deseja passar a utilizar a `ActionBar` . Para isso, declare a dependência da biblioteca de suporte no arquivo `build.gradle` do módulo da sua aplicação, dessa forma:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.0'
}
```

Para aplicar na prática o conteúdo deste capítulo e dos próximos, iniciaremos o desenvolvimento de um aplicativo que será um catálogo de produtos. Nosso catálogo terá como objetivo exibir um conjunto de produtos que o usuário poderá comprar ou encomendar. Você pode definir a temática dele; pode ser, por exemplo, um catálogo de veículos e acessórios, um de vestuário, de eletrônicos ou do que você desejar. A temática que mostrarei aqui será de um catálogo de livros da Casa do Código.

Crie uma nova aplicação Android com o nome **Catalog** e pacote `com.codecrushing.catalog` , usando o assistente do Android

Studio mencionado anteriormente. Utilizaremos os arquivos gerados pela IDE para compreender a configuração inicial de uma `ActionBar`. Abra a classe `MainActivity` e repare que já existe uma diferença na classe que ela estende. Para suportar a barra de ações, a sua `Activity` deve passar a estender `ActionBarActivity`. Observe também que a `ActionBarActivity` pertence a um pacote específico da `v7-appcompat`. O trecho de código a seguir destaca essas diferenças:

```
import android.support.v7.app.ActionBarActivity;
import android.support.v7.app.ActionBar;
//outros imports foram suprimidos

public class MainActivity extends ActionBarActivity {
    //restante do código suprimido
}
```

Vale a pena frisar que é importante estar atento aos `import`s corretos quando se usa a biblioteca de suporte. Geralmente, as classes que você precisará estão no pacote `android.support.<versão>` (como v7), ou tem o sufixo `Compat` no nome. Se um `import` for realizado incorretamente, o aplicativo deixará de funcionar nas versões mais antigas. Na classe `MainActivity`, também tem códigos referentes ao uso de `Fragments`.

Quando a `ActionBar` foi introduzida no Android, a ideia foi reaproveitar a infraestrutura de menus. Portanto, podemos fazer a definição dos itens de uma `ActionBar` em um arquivo de definição de menu. No projeto que criamos, já existe um arquivo inicial em `res/menu/main.xml`, com o seguinte conteúdo:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.codecrushing.catalog.MainActivity" >

    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="@string/action_settings"
```

```
        app:showAsAction="never"/>

</menu>
```

Basicamente, é uma definição de item de menu com a diferença da declaração do atributo `showAsAction` . O valor `never` indica que este item nunca deve ser exibido como ação na `ActionBar` e deve sempre ir para o *action overflow*; ou seja, ela não será mostrada nem como ícone, nem como texto na `ActionBar` , apenas como submenu. O atributo `showAsAction` está definido na biblioteca de suporte, cujos recursos são referenciados pelo namespace `xmlns:app="http://schemas.android.com/apk/res-auto"` . Esse namespace foi incluído recentemente e facilita a utilização de recursos da própria aplicação, como componentes customizados, e também de bibliotecas de terceiros.

Na Activity criada, o método `onCreateOptionsMenu` adiciona os itens definidos no `main.xml` na `ActionBar` . Para tratar a escolha realizada pelo usuário, sobrescrevemos o método `onOptionsItemSelected` e verificamos se o `MenuItem` informado corresponde à determinada ação. O trecho de código a seguir complementa a explicação:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

O último passo para colocar uma `ActionBar` na sua aplicação é aplicar um tema do Android que possua esse recurso. O assistente

de criação do projeto já fez isto por nós; vamos verificar o resultado. No arquivo `AndroidManifest.xml` existente no projeto criado e listado a seguir, repare que existe a definição de um tema por meio do atributo `android:theme` da tag `application` . Nesse caso, o tema foi aplicado para todo o aplicativo; no entanto, é possível configurar temas por `Activity` .

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.codecrushing.catalog"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="22" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.codecrushing.catalog.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
              <action android:name="android.intent.action.MAIN"/>
              <category
                android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

O valor do atributo `android:theme` é uma referência para um estilo com nome de `AppTheme` . Abra o arquivo `res/values/styles.xml` e observe que já existe uma hierarquia de estilos previamente definida. Isso é para facilitar a customização dos temas. Neste momento, o que interessa saber é que o tema que será aplicado em todo o aplicativo, é o seguinte:

```
<style name="AppBaseTheme"
```

```
        parent="Theme.AppCompat.Light.DarkActionBar">
</style>
```

O tema `Theme.AppCompat.Light.DarkActionBar` define cores mais claras e também inclui a `ActionBar` . Poderíamos usá-lo diretamente no `AndroidManifest.xml` , mas isso não nos deixaria espaço para fazer customizações. Agora que já dissecamos o projeto criado, execute-o em um emulador ou dispositivo de sua preferência. O resultado será algo semelhante ao demonstrado na próxima figura:

Figura 10.4: ActionBar inicial do catálogo

Agora que já temos a configuração inicial da nossa `ActionBar` , podemos incrementá-la com ações que farão parte do catálogo. Independentemente da sua temática, três ações que provavelmente devem existir são: o carrinho de compras, a pesquisa por produtos e um acesso rápido para finalizar o pedido (*checkout*); além, é claro, da ação de compartilhar. Uma ação simples de adicionar é a referente ao *checkout*. A ideia é que ela leve o usuário diretamente para uma `Activity` que trata do fechamento da compra e pagamento. Para isso, basta adicionar um novo item no arquivo `res/menu/main.xml` , da seguinte forma:

```
<item
        android:id="@+id/action_checkout"
        android:icon="@drawable/ic_action_cart"
        android:title="@string/action_checkout"
        app:showAsAction="ifRoom"/>
```

É desejável que a ação de finalizar a compra apareça na `ActionBar` sempre que possível, daí a escolha por utilizar o valor `ifRoom` no atributo `showAsAction` . Outros valores possíveis são: o `never` , que já vimos anteriormente; o `always` , que faz com que este item seja sempre mostrado; o `withText` , que exibe o título da ação; e o `collapseActionView` , que veremos a seguir. Apesar de ser possível, não é recomendável que itens sejam sempre fixos na `ActionBar` (`always`), pois, como existem diferentes tamanhos de tela, o layout da barra de ações pode ficar comprometido.

A ação selecionada pelo usuário deve ser tratada sobrescrevendo o método `onOptionsItemSelected` da `Activity` que contém a `ActionBar` . No projeto que criamos, já foi gerado um código inicial para tratar os seus eventos. Para simular o funcionamento das ações, vamos exibir um `Toast` quando o usuário selecionar uma ação. Na sua aplicação real, você deverá executar o procedimento mais adequado para a opção selecionada. Veja como ficou o código de exemplo:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if(id == R.id.action_checkout){
            Toast.makeText(this,
                           getString(R.string.action_checkout),
                           Toast.LENGTH_SHORT).show();
            return true;
    }
    return super.onOptionsItemSelected(item);
}
```

10.3 ADICIONE UMA PESQUISA RÁPIDA NA ACTIONBAR

A `ActionBar` possui um recurso chamado de *Action View*, que permite exibir widgets na barra, em resposta a uma ação, como por exemplo, possibilitar que o usuário faça a entrada de dados

diretamente na ActionBar . Isso é particularmente interessante para as ações de pesquisa, em que o usuário tem a possibilidade de digitar a informação desejada e executar a ação a partir da ActionBar . Para criar uma ação que se comporte dessa maneira, precisamos incluir mais um item nela, da seguinte maneira:

```
<item
    android:id="@+id/action_search"
    android:icon="@drawable/ic_action_search"
    android:title="@string/action_search"
    app:actionViewClass="android.support.v7.widget.SearchView"
    app:showAsAction="ifRoom|collapseActionView"/>
```

Android Asset Studio para criar ícones

Para facilitar a criação de ícones, você pode usar a ferramenta online *Android Asset Studio*, http://romannurik.github.io/AndroidAssetStudio. Com ela, é possível criar ícones a partir de uma figura ou texto, e já com os tamanhos específicos para cada resolução. Os ícones usados neste capítulo foram criados nessa ferramenta. O Google também disponibilizou um pacote de ícones do novo Material Design do Android 5.0 no site https://google.github.io/material-design-icons/.

No trecho de código anterior, utilizamos o atributo actionViewClass para determinar qual o tipo de widget deve ser apresentado na ActionBar , que, neste caso, será uma SearchView . Note que essa classe também faz parte da biblioteca de suporte.

Outra definição importante é como e quando o widget deve ser exibido. O showAsAction combina dois valores, o ifRoom e o collapseActionView , fazendo com que o ícone referente a ação

de pesquisa seja mostrado, caso haja espaço, e que o widget deve estar escondido. Quando o ícone de pesquisa for acionado, a `SearchView` será expandida e permitirá a entrada de dados. A figura a seguir mostra como ficou a `ActionBar` com as ações adicionadas, a `SearchView` expandida e também a *action overflow:*

Figura 10.5: ActionBar com ações e SearchView

A `SearchView` estabelece três interfaces para manipulação dos eventos gerados: a `OnCloseListener`, que notifica quando a *view* é fechada; a `OnQueryTextListener`, que define *callbacks* para os eventos relacionados ao texto informado pelo usuário; e, por último, a `OnSuggestionListener`, responsável por notificar os eventos relacionados a sugestões de pesquisa. Para tratar a pesquisa realizada pelo usuário, implementaremos a `OnQueryTextListener` na `MainActiviy`, conforme demonstra o código a seguir:

```
public class MainActivity extends ActionBarActivity
                            implements OnQueryTextListener{
    //demais códigos foram omitidos

    @Override
    public boolean onQueryTextChange(String novoTexto) {
        return false;
    }

    @Override
    public boolean onQueryTextSubmit(String texto) {
        return false;
    }
}
```

A interface `OnQueryTextListener` exige que dois métodos

sejam implementados. Neste momento, implementaremos o `onQueryTextSubmit`, que é disparado pela `SearchView` quando o usuário conclui a entrada de dados. Para efeitos didáticos, faremos uma busca simulada em nosso catálogo. Criaremos uma lista de produtos em memória e faremos a busca em cima dela, para encontrar produtos com o nome equivalente ao informado pelo usuário. Ao realizar a busca, o aplicativo vai apenas apresentar se foram ou não encontrados resultados para o texto informado. Para exibir esse resultado, utilizaremos as *views* que já existem no projeto. Analise os códigos a seguir:

```
<!-- fragment_main.xml -->
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">

    <TextView
        android:id="@+id/search_result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
         />

</RelativeLayout>

<!-- Arquivo strings.xml -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Catalog</string>
    <string name="action_settings">Settings</string>
    <string name="action_checkout">Checkout</string>
    <string name="action_my_cart">My Cart</string>
    <string name="action_search">Search</string>
    <string name="results_found">
        Results found: <b>%s</b>
    </string>
    <string name="results_not_found">
        Results not found: <b>%s</b>
    </string>
</resources>
```

```java
public class MainActivity extends ActionBarActivity
                            implements OnQueryTextListener{

    private List<String> products =
            Arrays.asList("android", "ios", "mobile",
                          "java", "agile");

    //demais códigos foram omitidos

    @Override
    public boolean onQueryTextSubmit(String text) {
        text = text.toLowerCase();
        TextView view = (TextView) getSupportFragmentManager()
                .findFragmentById(R.id.container).getView()
                .findViewById(R.id.search_result);

        for (String name : products) {
            if(name.contains(text)){
                String result =
                        getString(R.string.results_found, text);
                view.setText(result);
                return true;
            }
        }

        String result =
                    getString(R.string.results_not_found, text);
        view.setText(result);

        return true;
    }
}
```

O arquivo `fragment_main.xml` define a *view* que é apresentada quando se inicia o aplicativo. Até então, ela mostrava apenas um `TextView` com a frase "*Hello World!*". Alteramos o `id` desse componente para `search_result`, e também removemos o atributo `text` que terá seu valor atribuído programaticamente. O arquivo `strings.xml` também foi listado para demonstrar os textos que estão sendo usados até o momento no aplicativo.

Na `MainActivity`, acrescentamos uma lista simples de produtos em memória com alguns valores, que, neste exemplo, representam o título dos livros. No método `onQueryTextSubmit`,

convertemos o texto digitado para caixa baixa, a fim de evitar problemas na comparação. Em seguida, recuperamos o `TextView` existente na view `fragment_main.xml`, para exibir o resultado da pesquisa.

Depois de obter a `TextView`, percorremos a lista para verificar se algum livro possui parte do nome equivalente ao informado pelo usuário. Caso exista, mostraremos uma mensagem simples para ele, informando que resultados foram encontrados. Execute o aplicativo, realize uma pesquisa e veja o resultado!

10.4 AÇÕES CUSTOMIZADAS COM ACTIONPROVIDER E SUBMENUS

Além das `ActionViews`, a `ActionBar` também oferece a possibilidade de se estender as suas funcionalidades, utilizando um `ActionProvider`, cuja função é tratar as operações de determinada ação. Ele, inclusive, possibilita a criação de submenus e a exibição de layouts personalizados, quando uma ação é selecionada pelo usuário. O `ActionProvider` é útil quando se quer disponibilizar uma ação que pode ser executada de diferentes maneiras. Um exemplo é a ação de compartilhar um link e uma descrição de um produto do catálogo.

Os usuários geralmente possuem instalados em seus dispositivos diversos aplicativos capazes de compartilhar informações com sua rede de amigos. Logo, podemos criar um `ActionProvider` que redirecionará a ação de compartilhar para algum desses aplicativos. Para a ação de compartilhar, o Android já disponibiliza uma implementação por meio da classe `ShareActionProvider`. Para usá-la, temos que primeiro adicionar um novo item na barra de ações, dessa forma:

```
<item
    android:id="@+id/action_share"
```

```
    android:icon="@drawable/abc_ic_menu_share_holo_light"
    android:title="@string/action_share"
    app:actionProviderClass=
    "android.support.v7.widget.ShareActionProvider"
    app:showAsAction="ifRoom"/>
```

Em seguida, precisamos configurar a `Intent` que será disparada quando essa ação for selecionada. No método `onCreateOptionsMenu` , devemos vincular um `ShareActionProvider` com o `MenuItem` correspondente à ação de compartilhar, e também informar uma `Intent` com as informações relacionadas ao compartilhamento. Veja o código:

```
private ShareActionProvider shareActionProvider;

@Override
public boolean onCreateOptionsMenu(Menu menu) {
   //códigos existentes foram omitidos

    MenuItem shareItem = menu.findItem(R.id.action_share);
    shareActionProvider = (ShareActionProvider)
            MenuItemCompat.getActionProvider(shareItem);
    shareActionProvider.setShareIntent(getShareIntent());

    return true;
}

private Intent getShareIntent() {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("text/html");
    intent.putExtra(Intent.EXTRA_TEXT, getInformation());
    return intent;
}

private String getInformation(){
    //get here any information you want to share,
    //possibly the description and link of
    //current product
    return "Information for sharing";
}
```

Um ponto importante de salientar é que, para implementar esse recurso, continuamos utilizando a biblioteca de compatibilidade por meio da classe `MenuItemCompat` . No método `getShareIntent` ,

definimos o conteúdo e o tipo do nosso compartilhamento. Ao selecionar a opção compartilhar, o aplicativo exibirá uma janela semelhante à mostrada na figura a seguir:

Figura 10.6: Ação Compatilhar com ShareActionProvider

Uma outra possibilidade de aprimorar a barra de ações é incluir opções que oferecem submenus. Considere que, em nosso catálogo, desejamos oferecer aos usuários a possibilidade de filtrar livros por assunto. Podemos incluir uma ação na `ActionBar` que, quando acionada, exibe uma lista de assuntos e o usuário pode marcar, por meio de uma `checkbox` , os que deseja filtrar, como mostra a figura a seguir:

Figura 10.7: Submenu com checkboxes

A definição de uma ação com submenu pode ser realizada diretamente nos arquivos de menu, adicionando um novo elemento `<menu>` a um `<item>`. Para que os itens desse submenu comportem-se como uma checkbox, é necessário agrupá-los e configurar o seu comportamento usando o elemento `<group>`. O código a seguir define o submenu que foi demonstrado na figura anterior.

```
<!-- res/menu/main.xml -->
<item
    android:id="@+id/action_filter"
    android:icon="@drawable/ic_action_filter"
    android:title="@string/action_filter"
    app:showAsAction="ifRoom">
    <menu>
        <group android:checkableBehavior="all" >
            <item
                android:id="@+id/action_filter_android"
                android:title="@string/filter_android"/>
            <item
                android:id="@+id/action_filter_mobile"
                android:title="@string/filter_mobile"/>
            <item
```

```
                android:id="@+id/action_filter_java"
                android:title="@string/filter_java"/>
        </group>
    </menu>
</item>
```

Repare que o item referente a ação filtrar possui agora um elemento menu, e este, por sua vez, possui diversos itens. Além disso, o grupo define, por meio do atributo `checkableBehavior`, que todos os itens podem ser selecionados. É essa configuração que determina que os itens do submenu são checkboxes. Outra opção possível para o atributo `checkableBehavior` é a `single`, que permite a seleção de apenas um item do submenu. Outro ponto importante é que só é possível ter um nível de submenu; ou seja, um item do submenu não pode abrigar um novo submenu.

Para tratar a opção do submenu que foi selecionada pelo usuário, também utilizaremos o `onCreateOptionsMenu`. Nesse método, devemos, além de executar o código pertinente à ação, também precisaremos assinalar que o item do menu foi selecionado. Dessa forma, a checkbox escolhida aparecerá selecionada. Confira no código a seguir:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if(id == R.id.action_filter_android) {
        //search for books in Android category
        item.setChecked(true);
        return true;
    }
    //....
}
```

10.5 CONCLUSÃO

Neste capítulo, aprendemos como utilizar as bibliotecas de compatibilidade para atender a um número maior de versões de Android, como também usar, por meio dela, recursos mais recentes

como o `ActionBar` . Vimos como incluir ações na `ActionBar` , recursos de pesquisa rápida e compartilhamento, e também o uso de submenus. No próximo capítulo, veremos outros aspectos importantes da `ActionBar` relacionados à navegação.

Capítulo 11

DEFINA UM ESQUEMA DE NAVEGAÇÃO

Um dos pontos chave para o engajamento do usuário é o esquema de navegação entre telas e funcionalidades, implementado pelo aplicativo. Grandes mudanças nessa área foram introduzidas a partir do Android 3.0, bem como o Google começou a demonstrar mais preocupação com a experiência do usuário. Ele estabeleceu diretrizes (*guidelines*) de design com recomendações importantes sobre a estrutura da aplicação e sua respectiva navegação, além de tipografia, iconografia, estilo de escrita e muito mais.

Essas diretrizes estão disponíveis em http://developer.android.com/design/index.html e sua leitura é obrigatória para levar sua aplicação para o próximo nível. Neste capítulo, trataremos as principais diretrizes de navegação e incorporaremos alguns recursos de navegação ao nosso catálogo de exemplo.

Diversos são os tipos de aplicativo e cada um deles possui características próprias. Por conta disso, não existe uma regra única de como implementar um esquema de navegação para o seu aplicativo; o que existe é um conjunto de recomendações que são úteis em determinados contextos. Ao se projetar como o usuário navegará entre as funcionalidades do aplicativo, é necessário levar em conta quantos níveis de visualização e de detalhe vão existir. Esses níveis podem ser classificados em visões de alto nível (*top level*

views), de categoria (*category views*) e de detalhe e edição (*detail/edit views*).

As visões de alto nível representam as principais informações do aplicativo e são a partir delas que, geralmente, os usuários iniciam a utilização. Já as de categoria servem para ligar as de alto nível às visões de detalhes, caso existam muitas visões que precisam ser agrupadas. Por último, as visões de detalhe e edição compõem o nível mais granular da hierarquia de navegação, e é onde tipicamente realiza-se alguma operação envolvendo dados e informações do aplicativo.

A recomendação do Google é que se tenha atenção especial para as visões de alto nível, já que elas são as telas de entrada para o seu aplicativo e devem proporcionar uma boa experiência de uso tanto para os novos usuários quanto para os frequentes. Também é sugerido que, nessas visões, o foco esteja no conteúdo do aplicativo, permitindo ao usuário facilmente realizar operações. Portanto, deve-se evitar visões que sirvam simplesmente como ponto de partida para a navegação. O uso de `ActionBar` em todas as telas também é encorajado.

Geralmente, os aplicativos possuem mais de uma visão de alto nível, sendo necessário definir uma forma fácil para que o usuário alterne entre as demais visões existentes. Para essas situações, existem três maneiras recomendadas de implementar a navegação: utilizando abas fixas (*fixed tabs*), *spinners* ou o *Navigation Drawer*. Evite combinar esses esquemas de navegação; a sugestão é escolher aquele que seja mais adequado ao seu aplicativo.

11.1 ABAS E SWIPE VIEWS

O uso de abas fixas permite que o usuário perceba facilmente a existência de outras visões disponíveis no aplicativo. É importante

ressaltar que sua quantidade deve ser fixa e limitada a, no máximo, três. Além disso, é necessário que a troca entre as abas possa ser realizada pelo usuário, ao arrastar o conteúdo da direita para esquerda (e vice-versa), em um movimento de *swipe*. Devem-se utilizar abas fixas para navegação quando se espera que os usuários alternem frequentemente entre elas, ou se deseja deixar clara a existência das demais visões de alto nível a ele.

Um bom exemplo de uso de abas fixas no nosso aplicativo de catálogo é criar visões de alto nível para exibir os itens mais vendidos, os lançamentos e os com desconto. Para deixar os códigos de exemplo separados e facilitar o entendimento, vamos criar uma nova `Activity` e novas classes para implementar a navegação por abas. A primeira coisa que vamos fazer é o mecanismo de *swipe views*, utilizando um `ViewPager` disponível na biblioteca de suporte. O `ViewPager` é um widget de layout, no qual permite navegação lateral e cada *view* filha é uma página separada.

Crie uma nova classe chamada `TabActivity` e faça-a estender `ActionBarActivity`. O layout dessa `activity` será apenas o `ViewPager`, e seu respectivo arquivo deverá ter o nome de `activity_tab.xml`. Veja como ficou:

```
public class TabActivity extends ActionBarActivity{

    private ViewPager viewPager;
    private ActionBar actionBar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab);

        viewPager = (ViewPager) findViewById(R.id.pager);
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

O ViewPager por si só não faz nada sozinho. Ele precisa que seja fornecido um PagerAdapter que é o responsável por determinar qual tela deve ser exibida em uma determinada posição. No caso do ViewPager , essas telas precisam ser Fragments . Então, o que teremos na prática é que, para cada posição do ViewPager , teremos um Fragment representando uma das abas do aplicativo. Para criar um PagerAdapter , basta estender um dos adaptadores já existentes.

No nosso caso, como temos um número conhecido e limitado de abas, estenderemos a classe FragmentPagerAdapter . Para os casos em que o número de abas é maior, deve-se optar por estender FragmentStatePagerAdapter , que lida melhor com o uso de memória. Criaremos agora um PagerAdapter bem simples, implementando dois métodos: um que devolve a quantidade de itens existentes (apenas 3, representado as abas existentes), e outro que deve retornar o Fragment que está na posição informada como parâmetro.

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class TabPagerAdapter extends FragmentPagerAdapter{

    private static final int NUM_TABS = 3;

    public TabPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public Fragment getItem(int position) {
        return new SimpleFragment(position);
    }

    @Override
```

```
    public int getCount() {
        return NUM_TABS;
    }
}
```

Repare no método `getItem` que retorna um `SimpleFragment`. Esta é uma classe que criaremos para representar nossa aba de forma simples. Este `Fragment` receberá a informação de sua posição e a exibirá na tela para verificarmos o funcionamento. Os códigos da `SimpleFragment` e do seu respectivo layout pode ver visto a seguir:

```
public class SimpleFragment extends Fragment {

    private int position;

    public SimpleFragment(){}

    public SimpleFragment(int position) {
        this.position = position;
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {

        View rootView = inflater.inflate(R.layout.fragment_tab,
                                         container, false);

        TextView textView =
                    (TextView) rootView.findViewById(R.id.item);

        textView.setText("Tab " + position);

        return rootView;
    }
}
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center|center_horizontal" >
```

```
    <TextView
        android:id="@+id/item"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="60sp" />

</LinearLayout>
```

Para finalizar a implementação do mecanismo de *swipe views*, precisamos configurar o `ViewPager` e seu respectivo `PagerAdapter` na nossa atividade, da seguinte forma:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_tab);

    viewPager = (ViewPager) findViewById(R.id.pager);

    FragmentManager fm = getSupportFragmentManager();
    TabPagerAdapter pagerAdapter = new TabPagerAdapter(fm);
    viewPager.setAdapter(pagerAdapter);
}
```

Para facilitar a execução do teste, altere no `AndroidManifest.xml` para que a `TabActivity` seja a atividade a ser iniciada quando o aplicativo for aberto. Ao executá-lo, você obterá um resultado semelhante ao exibido na figura a seguir:

Figura 11.1: ViewPager e swipe views

Note que, apesar de termos a possibilidade de trocar de tela usando gestos, ainda não temos de fato as abas fixas que vão auxiliar o usuário na navegação. Acrescentá-las será o nosso próximo passo. O que precisamos fazer é integrar o funcionamento do `ViewPager` ao mecanismo de abas da `ActionBar` . Para tal, vamos configurar o modo de navegação da nossa barra de ações e também adicionar programaticamente as três abas que desejamos, da seguinte forma:

```
SimpleTabListener listener = new SimpleTabListener(viewPager);
actionBar.addTab(
        actionBar.newTab()
                 .setText(R.string.top_sellers)
                 .setTabListener(listener));

actionBar.addTab(
        actionBar.newTab()
                 .setText(R.string.new_arrivals)
                 .setTabListener(listener));

actionBar.addTab(
        actionBar.newTab()
                 .setText(R.string.off)
                 .setTabListener(listener));
```

Quando utilizamos abas na barra de ações, é necessário fornecer um `TabListener` que será o responsável por responder aos eventos relacionados às suas mudanças. A nossa implementação do `TabListener` deverá receber o `ViewPager` e, quando alguma aba for selecionada, alteraremos a posição do `ViewPager` para refletir a seleção realizada pelo usuário. Temos a seguir o código do `SimpleTabListener` :

```
public class SimpleTabListener implements TabListener {

    private ViewPager viewPager;

    public SimpleTabListener(ViewPager viewPager) {
        this.viewPager = viewPager;
    }

    @Override
    public void onTabSelected(Tab tab, FragmentTransaction ft){
```

```
        viewPager.setCurrentItem(tab.getPosition());
    }

    @Override
    public void onTabUnselected(Tab tab,FragmentTransaction ft){
    }

    @Override
    public void onTabReselected(Tab tab,FragmentTransaction ft){
    }
}
```

Ainda há mais uma integração a ser feita, que é fazer com o que o `ViewPager` atualize a aba atual da `ActionBar` quando o usuário desliza entre as páginas. Isso é feito criando um *listener* para lidar com a troca de páginas, da seguinte forma:

```
viewPager.setOnPageChangeListener(
        new ViewPager.SimpleOnPageChangeListener(){
            @Override
            public void onPageSelected(int position) {
                actionBar.setSelectedNavigationItem(position);
            }
});
```

Ao executar o aplicativo, você terá um resultado semelhante ao apresentado na figura seguinte. O código a seguir mostra como ficou a implementação da `TabActivity` .

Figura 11.2: Navegação utilizando abas

```
public class TabActivity extends ActionBarActivity{

    private ViewPager viewPager;
    private ActionBar actionBar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab);

        viewPager = (ViewPager) findViewById(R.id.pager);

        FragmentManager fm = getSupportFragmentManager();
        TabPagerAdapter pagerAdapter = new TabPagerAdapter(fm);
        viewPager.setAdapter(pagerAdapter);

        actionBar = getSupportActionBar();
        actionBar
            .setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

        SimpleTabListener listener =
                        new SimpleTabListener(viewPager);

        actionBar.addTab(
                actionBar.newTab()
                         .setText(R.string.top_sellers)
                         .setTabListener(listener));

        actionBar.addTab(
                actionBar.newTab()
                         .setText(R.string.new_arrivals)
                         .setTabListener(listener));

        actionBar.addTab(
                actionBar.newTab()
                         .setText(R.string.off)
                         .setTabListener(listener));

        viewPager.setOnPageChangeListener(
                new ViewPager.SimpleOnPageChangeListener(){
                    @Override
                    public void onPageSelected(int position) {
                        actionBar
                            .setSelectedNavigationItem(position);
                    }
        });
    }
}
```

11.2 NAVIGATION DRAWER

O *Navigation Drawer* foi uma das principais adições em termos de navegação introduzidas recentemente no Android. Esse recurso é basicamente um painel que apresenta opções de navegação para o usuário, e abre-se quando se desliza (*swiping*) da borda esquerda da tela para a direita, ou se seleciona o ícone da aplicação na `ActionBar` .

Seus propósitos são agrupar um grande número de visões disponíveis e ser acessível a partir de qualquer lugar do seu aplicativo. Estruturas de navegação complexas e com muitas opções podem ser implementadas usando o *Navigation Drawer*. Podemos utilizá-lo quando necessitamos de acesso direto a visões que não estão diretamente relacionadas, e também quando desejamos acessar diretamente visões de detalhes.

As situações mais comuns nas quais se deve empregar o *Navigation Drawer* são: quando o seu aplicativo possui mais de três visões de alto nível (impossibilitando o uso de abas fixas); quando existe uma navegação cruzada entre visões de nível mais baixo na hierarquia de navegação; e quando essa hierarquia é composta por vários níveis e desdobramentos.

Para exemplificar seu uso no aplicativo de catálogo, vamos criar uma nova `activity` que permita a navegação entre categorias de livro disponíveis. Crie uma nova classe chamada de `DrawerActivity` e um arquivo de layout com nome de `activity_drawer.xml` .

Para utilizar o *Navigation Drawer*, precisamos incluir um widget chamado `DrawerLayout` , que é a base para o layout do painel e está disponível na biblioteca de suporte. O *drawer* é composto basicamente por dois elementos, sendo que o primeiro deles precisa ser necessariamente um elemento de layout que representará o

conteúdo da sua `activity` ; e o segundo ser uma `ListView` que exibirá as opções de navegação disponíveis. O código a seguir mostra como deve ficar o arquivo de layout `activity_drawer.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <ListView
        android:id="@+id/nav_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:background="#fff"
        android:choiceMode="singleChoice" />

</android.support.v4.widget.DrawerLayout>
```

O conteúdo principal representado pelo `FrameLayout` deve ser o primeiro elemento, pois a ordem em que os elementos aparecem no XML implica a sua posição no eixo Z, que indica a profundidade. A `ListView` , declarada como segundo elemento, fará com que o `drawer` seja renderizado na frente do conteúdo da *view* principal. Repare que o `FrameLayout` foi configurado para ocupar todo o espaço disponível na horizontal e na vertical, enquanto a `ListView` teve a largura reduzida. O objetivo disso é permitir que o conteúdo principal permaneça parcialmente visível.

Outra configuração importante é do atributo `layout_gravity` com o valor `start` . Ele permite o posicionamento correto do *drawer*, mesmo quando o dispositivo do usuário estiver com um idioma com leitura da direita para a esquerda.

Como as opções do nosso `drawer` serão exibidas por uma `ListView`, vamos criar um `array` de `strings`, com as suas respectivas descrições, e usar um `ArrayAdapter`. A declaração do `array` de opções e o código inicial da `DrawerActivity` ficarão assim:

```
<!-- strings.xml -->
<string-array name="drawer_options">
    <item>Front End</item>
    <item>Games</item>
    <item>Java</item>
    <item>Mobile</item>
    <item>Web Development</item>
</string-array>
```

```
public class DrawerActivity extends ActionBarActivity{

    private ListView drawerList;
    private DrawerLayout drawerLayout;
    private String[] options;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drawer);

        options = getResources()
                    .getStringArray(R.array.drawer_options);

        drawerLayout =
            (DrawerLayout) findViewById(R.id.drawer_layout);
        drawerList = (ListView) findViewById(R.id.nav_drawer);

        drawerList.setAdapter(new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, options));
    }
}
```

Ao executar o aplicativo e abrir o *drawer*, teremos um resultado semelhante ao exibido na figura mais adiante. Neste momento, temos apenas o mínimo de layout e o funcionamento básico de abrir e fechar o *drawer*.

Agora, vamos programar as respostas aos eventos de seleção do

usuário. Ao escolher uma opção do *drawer*, o que faremos é substituir o conteúdo principal, representado pelo `FrameLayout` , pelo conteúdo equivalente ao selecionado pelo usuário.

Em nosso exemplo, faremos a substituição do `FrameLayout` por um `Fragment` com o conteúdo desejado. No entanto, é possível também que, em vez disso, você opte por abrir outra `activity` . A melhor forma de fazer dependerá de como você vai estruturar a sua aplicação. Veja a seguir como ficou a implementação do *listener*:

```
public class DrawerActivity extends ActionBarActivity
                            //handle user click
                            implements OnItemClickListener{

    //new attribute
    private CharSequence title;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        drawerList.setOnItemClickListener(this);
    }

    @Override
    public void onItemClick(AdapterView<?> adapter, View view,
                            int position, long id) {

        Fragment fragment = new SimpleFragment(position);

        getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.content_frame, fragment)
                .commit();

        drawerList.setItemChecked(position, true);
        title = options[position];
        setTitle(title);
        drawerLayout.closeDrawer(drawerList);
    }
}
```

Figura 11.3: Navegação utilizando abas

No método `onCreate`, adicionamos uma linha para vincular a `ListView` ao novo *listener*, cuja implementação acontece no método `onItemClick`. Nesse método, criamos um novo `SimpleFragment` já usado anteriormente, e fizemos a substituição do `FrameLayout` pelo `Fragment` recém-criado. Além disso, marcamos na `ListView` qual item foi selecionado, usando o método `setItemChecked`, e alteramos o título da `ActionBar` para refletir a opção selecionada pelo usuário. Por último, fechamos o *drawer* invocando o método `closeDrawer`.

Ao executar o aplicativo novamente, você perceberá que já temos a seleção de opções funcionando! Ainda faltam os últimos ajustes. Quando utilizamos um *drawer* e uma `ActionBar`, é recomendado que as ações contextuais da `ActionBar` sejam ocultadas, quando o *drawer* for aberto; e tornem-se visíveis novamente, quando for fechado. Além disso, é importante as telas que possuem um *drawer* tenham uma indicação visual na barra de ações para indicar a sua existência.

Você já deve ter percebido em alguns aplicativos a existência de um ícone composto de três listras no canto superior esquerdo,

indicando a existência do *Navigation Drawer*. Também é preciso permitir que o usuário abra e feche o *drawer* a partir desse ícone. Outro detalhe importante é que, quando ele for aberto, o título da barra de ações deve mudar, geralmente, para o nome da aplicação e não mais ter o título da tela que estava sendo exibida.

Para tratar os eventos de abertura e fechamento do *drawer*, vamos usar uma classe base já existente, chamada de `ActionBarDrawerToggle`. Nela, implementaremos dois métodos: um para tratar da abertura, que atualizará o título da barra de ações para o nome da aplicação; e outro do fechamento que alterará o título de volta para ao que corresponde a *view* exibida. Analise o código a seguir e veja como ficou o método `onCreate` da `DrawerActivity` com essa implementação:

```
//new attributes
private ActionBarDrawerToggle drawerToggle;
private CharSequence drawerTitle;

@Override
protected void onCreate(Bundle savedInstanceState) {

    ...

    title = drawerTitle = getTitle();

    drawerToggle = new ActionBarDrawerToggle(this, drawerLayout,
                    R.drawable.ic_drawer, R.string.drawer_open,
                    R.string.drawer_close) {

            public void onDrawerClosed(View view) {
                super.onDrawerClosed(view);
                getSupportActionBar().setTitle(title);
                supportInvalidateOptionsMenu();
            }

            public void onDrawerOpened(View drawerView) {
                super.onDrawerOpened(drawerView);
                getSupportActionBar().setTitle(drawerTitle);
                supportInvalidateOptionsMenu();
            }
    };
```

```
    drawerLayout.setDrawerListener(drawerToggle);
}
```

Na criação do `ActionBarDrawerToggle` , precisamos fornecer, além do contexto e da referência para o widget do *drawer*, o ícone que será utilizado para sinalizar sua existência e duas `strings` descrevendo as ações de abrir e fechar para questões de acessibilidade. Na implementação do `onDrawerClosed` e do `onDrawerOpened` , o título da barra de ações é alterado conforme explicado anteriormente e o método `supportInvalidateOptionsMenu` é invocado.

O objetivo disso é fazer com que as opções da barra de ação sejam reconstruídas nos dando a chance de ocultar ou exibir as ações contextuais da `ActionBar` . Para isso, sobrescrevemos o método `onPrepareOptionsMenu` , verificamos o estado do *drawer* (aberto ou fechado) e optamos por ocultar ou exibir uma determinada ação. O trecho de código a seguir exemplifica como seria esconder ou mostrar a ação de pesquisar na barra de ações:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    boolean drawerOpen = drawerLayout.isDrawerOpen(drawerList);
    menu.findItem(R.id.action_search).setVisible(!drawerOpen);
    return super.onPrepareOptionsMenu(menu);
}
```

Continuando nossa implementação do *Navigation Drawer*, é preciso configurar a `ActionBar` para exibir o ícone do *drawer* e também capturar os eventos de abrir e fechar provenientes da `ActionBar` . Para configurá-la, basta invocar dois métodos no `onCreate` , indicando que seu ícone funciona como um botão e repassando o evento para o `ActionBarDrawerToggle` , de modo que ele dispare o funcionamento do *drawer*. Isso deve ser feito ao sobrescrever o método `onOptionsItemSelected` , que é o mesmo responsável por tratar os eventos de seleção na `ActionBar` .

Confira como o código ficou a seguir:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    getSupportActionBar().setHomeButtonEnabled(true);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (drawerToggle.onOptionsItemSelected(item)) {
      return true;
    }
    //handle your options
    return super.onOptionsItemSelected(item);
}
```

O último passo é garantir que o *Navigation Drawer* e a `ActionBar` permaneçam sincronizados após uma mudança de configuração do dispositivo, como uma rotação de tela, por exemplo. Para isso, utilizamos o código a seguir que basicamente invoca métodos do `ActionBarDrawerToggle` durante algumas etapas de construção da `Activity` .

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
    super.onPostCreate(savedInstanceState);
    drawerToggle.syncState();
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    drawerToggle.onConfigurationChanged(newConfig);
}
```

Ao executar o aplicativo novamente, você terá uma implementação funcional do *Navigation Drawer*! Compare o seu resultado com a figura:

Figura 11.4: Navigation Drawer

11.3 SPINNERS

Outra forma de organizar a navegação entre visões de alto nível é usando `Spinners` na `ActionBar` . Eles funcionam como um menu no estilo *drop-down* e servem para alternar o conteúdo quando uma opção é selecionada.

Os `Spinners` , assim como as abas fixas, devem ser utilizados quando há um número pequeno de visões de alto nível disponíveis para o usuário. Eles também são uma boa opção quando se deseja aproveitar melhor o espaço, já que não exige uma barra específica além da `ActionBar` .

No entanto, uma recomendação importante é que as opções do `Spinners` devem ser correlacionadas, seja porque o usuário está navegando entre visões em cima do mesmo conteúdo — como, por exemplo, eventos da agenda, visualizados por dia, mês etc. —, ou porque está navegando entre visões com conteúdo do mesmo tipo — como e-mails da conta A e da conta B.

Para praticar esse tipo de navegação, usaremos um `Spinner` na barra de ação para navegar entre os itens mais vendidos, os recém-adicionados e os com desconto. Ao selecionar um item da lista, vamos alterar o `Fragment` que é exibido.

A implementação desse tipo de navegação é relativamente simples; precisaremos apenas construir um SpinnerAdapter para exibir as opções disponíveis, configurar o tipo de navegação na ActionBar e atribuir um *callback* para tratar os eventos de seleção. Crie uma nova classe chamada SpinnerActivity , e utilize o arquivo de layout activity_main.xml já existente no projeto, cujo código é apresentado a seguir:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:ignore="MergeRootFrame" />
```

Usaremos o FrameLayout apenas como marcação para que possamos substituí-lo pelo Fragment correspondente quando o usuário selecionar uma opção. Assim como fizemos anteriormente, vamos criar as opções disponíveis como um array e, a partir dele, criaremos um SpinnerAdapter . O código a seguir exibe sua declaração:

```
<!-- strings.xml -->
<string-array name="spinner_options">
    <item>Top Seller</item>
    <item>New Arrivals</item>
    <item>10% Off</item>
</string-array>
```

Como estamos utilizando uma ActionBar com cor escura, vamos criar um layout específico para o item spinner e fazer com que a cor do texto seja branca. O código apresentado na sequência mostra o conteúdo do arquivo dropdown_item.xml com essa configuração:

```
<CheckedTextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:gravity="center"
    android:layout_height="40dp"
```

```
    android:singleLine="true"
    android:textColor="#fff" />
```

A criação do `SpinnerAdapter` será feita usando o método utilitário `createFromResource` da classe `ArrayAdapter`, que recebe como parâmetro uma referência para o `array` de opções e o layout que será utilizado para cada item. O código a seguir demonstra a criação do `adapter`:

```
public class SpinnerActivity extends ActionBarActivity

    private SpinnerAdapter adapter;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        adapter = ArrayAdapter.createFromResource(this,
                    R.array.spinner_options,
                    R.layout.dropdown_item);
    }
}
```

Agora resta apenas configurar o tipo de navegação da `ActionBar` e o *callback* que tratará os eventos de seleção. Faremos nossa `activity` implementar `OnNavigationListener` e implementaremos o método `onNavigationItemSelected`, que simplesmente vai trocar o `FrameLayout` por um `SimpleFragment` (já usado anteriormente), correspondente à posição selecionada. Veja como ficou o código final da `SpinnerActivity` e o aplicativo na figura adiante:

```
public class SpinnerActivity extends ActionBarActivity
            implements OnNavigationListener {

    private SpinnerAdapter adapter;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        adapter = ArrayAdapter.createFromResource(this,
                    R.array.spinner_options,
```

```
                    R.layout.dropdown_item);

        getSupportActionBar()
            .setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
        getSupportActionBar()
            .setListNavigationCallbacks(adapter, this);
    }

    @Override
    public boolean onNavigationItemSelected(int position,
                    long itemId) {

        Fragment fragment = new SimpleFragment(position);

        getSupportFragmentManager()
            .beginTransaction()
            .replace(R.id.container, fragment)
            .commit();

        return true;
    }
}
```

Figura 11.5: Navegação utilizando spinner

11.4 RECOMENDAÇÕES PARA ACTIONBAR

Uma das questões centrais em termos de estrutura da aplicação e navegação é a utilização de uma `ActionBar` . Apesar de já termos tratado desse assunto no capítulo *Desenvolva para todas as versões do Android*, aqui vale ressaltar novamente a sua importância e analisar seu uso agora sob o aspecto do design e da usabilidade.

Outro ponto importante é como ajustar o conteúdo da aplicação, inclusive a `ActionBar` , para os diferentes tamanhos de tela e também lidar com sua rotação. Para essas situações, existe a possibilidade de distribuir as ações e recursos de navegação em outras regiões da tela que não seja na parte superior, onde geralmente se localiza a `ActionBar` . A figura a seguir ilustra os locais que podem ser usados:

Figura 11.6: Regiões para ActionBar. Fonte: http://developer.android.com

1. **Barra principal —** está localizada no topo da tela e pode incluir o ícone do aplicativo e uma informação textual de onde o usuário se encontra. Se o usuário puder navegar de volta para uma tela de nível superior, a `ActionBar` deve possuir o indicador de voltar, representado por um ícone semelhante a "<".

2. **Barra superior —** este local dever ser utilizado para incluir recursos de navegação, como abas e *spinners*, permitindo que o usuário navegue entre as principais funcionalidades do aplicativo.

3. **Barra inferior —** quando se divide a `ActionBar`, este local deve ser usado para abrigar as ações e também o *action overflow*.

Para se dividir a `ActionBar`, é necessário incluir essa configuração no `AndroidManifest.xml` para toda a aplicação ou apenas para as atividades que terão a barra de ações dividida. Essa configuração é realizada por meio do atributo `uiOptions`, que pode ser colocado na tag `<application>` e também na `<activity>`. Veja exemplos dessas duas formas de configurar nos códigos a seguir:

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:uiOptions="splitActionBarWhenNarrow" >
    ...
</application>

<activity
    android:name="com.codeslashers.catalog.MainActivity"
    android:label="@string/app_name"
    android:uiOptions="splitActionBarWhenNarrow">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category
            android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

O atributo `uiOptions` só é reconhecido a partir da API 14 do Android. Para utilizar esse recurso em versões anteriores, por meio das bibliotecas de compatibilidade, é necessário configurar as atividades usando metainformações, da seguinte maneira:

```xml
<activity
    android:name="com.codeslashers.catalog.MainActivity"
    android:label="@string/app_name">
    <meta-data
        android:name="android.support.UI_OPTIONS"
        android:value="splitActionBarWhenNarrow" />
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
            android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

CAPÍTULO 12

PUBLICAÇÃO NO GOOGLE PLAY

Chegou a hora de publicar a sua app para o mundo! Para realizar o registro como desenvolvedor no Google Play, você precisará de uma conta do Google e de um cartão de crédito internacional para realizar o pagamento da taxa de U$ 25,00. O processo é simples e serve tanto para pessoa física como para empresas. Veja na seção a seguir como criar a sua conta e realizar a publicação de uma app.

12.1 PREPARE A APLICAÇÃO

Antes de mais nada, é necessário preparar a sua aplicação para que ela possa ser publicada e posteriormente atualizada no Google Play. Os primeiros itens que devemos conferir é no `AndroidManifest.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.casadocodigo.boaviagem"
    android:versionCode="1"
    android:versionName="1.0" >
<!-- demais itens -->
```

O nome do pacote, especificado no atributo `package` (linha 3), deve ser único para identificar a sua aplicação no Google Play; por isso é comum utilizar algo semelhante a um domínio de internet.

Uma vez publicada, não será mais possível alterar esse nome.

O atributo `versionCode` é outro atributo importante que identifica a versão atual da aplicação. Ele é usado para verificar se há atualizações a serem realizadas no aplicativo. A cada nova versão esse número deve ser incrementado.

Já o atributo `versionName` (linha 4) é de uso livre para o desenvolvedor nomear a versão da forma que quiser. Ele é exibido no Google Play e também no próprio dispositivo, enquanto que o `versionCode` (linha 5) é invisível para o usuário.

Dando continuidade à preparação, o Android exige que todos os aplicativos sejam assinados digitalmente, como uma forma de identificar o autor da aplicação. Durante o desenvolvimento, o Android Studio já cuida da geração e assinatura do `apk` sem termos que nos preocupar. Para a publicação no Google Play, criaremos um certificado digital autoassinado, para realizar a assinatura do `apk`.

Para que as atualizações do aplicativo possam ser aplicadas de forma transparente para o usuário, é importante sempre assinar a aplicação com o mesmo certificado. Quando o Android atualiza um aplicativo, é verificado se os certificados da versão nova coincidem com aqueles da versão instalada. Se forem idênticos, então a atualização é realizada. Caso contrário, não será possível realizá-la. A seguir, veja como exportar, gerar o certificado e assinar o `apk` para publicação.

Figura 12.1: Exportação

Como vemos nessa figura, acesse o menu `Build > Generated Signed Apk` . Na tela seguinte, mostrada na figura a seguir, selecione a opção `Create new` para criar uma nova `keystore` , pois é a primeira vez que estamos realizando esse processo. Na próxima exportação, você poderá escolher a opção de utilizar um `keystore` existente. Informe uma senha segura e sua confirmação.

Figura 12.2: Exportação — Passo 1

Figura 12.3: Exportação — Passo 2

Na próxima tela, ilustrada nessa última figura, informaremos os dados da chave. É recomendado que a validade do certificado seja superior a 25 anos. Informe novamente uma senha segura, um `alias` e a identificação do autor da aplicação. Os demais campos são opcionais.

Figura 12.4: Exportação — Passo 3

Para finalizar a exportação, na próxima tela selecione o local de destino do arquivo `apk` assinado e clique em `Finish`, como mostra a figura anterior.

Agora a sua aplicação está pronta para ser publicada!

12.2 CRIE UMA CONTA DE DESENVOLVEDOR

Figura 12.5: Android Developer Console" label=play1

Acesse o site https://play.google.com/apps/publish/ e faça o login usando a sua conta Google (figura anterior). Uma tela semelhante à exibida na figura a seguir será apresentada para o preenchimento dos dados básicos.

Figura 12.6: Informações básicas

Em seguida, aceite o contrato de distribuição como ilustra a figura:

Figura 12.7: Contrato de distribuição

Logo após, é necessário realizar o pagamento da taxa com um cartão de crédito internacional, utilizando o serviço Google Wallet, como mostram as imagens:

Figura 12.8: Pagamento da taxa — Tela 1

Figura 12.9: Pagamento da taxa — Tela 2

Após o pagamento ter sido realizado com sucesso, você terá acesso ao console do desenvolvedor, mostrado na figura:.

Figura 12.10: Android Developer Console

Caso a sua aplicação não seja gratuita, será necessário criar uma conta de comerciante (*Merchant Account*) para que você possa receber os valores das vendas. Para criar esse tipo de conta, utilize a opção disponível no próprio site Android Developer Console e preencha as informações necessárias.

12.3 REALIZE A PUBLICAÇÃO

Para publicar a sua aplicação, no console do desenvolvedor (figura anterior) escolha a opção **Enviar aplicativo**. Uma tela será aberta para a realização do upload do arquivo `.apk` que foi previamente preparado, conforme mostra a primeira figura a seguir. Depois disso, será necessário preencher informações adicionais sobre o aplicativo, incluindo o ícone em alta resolução, capturas de

tela, imagens e textos de divulgação e a descrição do que a aplicação faz (segunda figura a seguir). Seja caprichoso na produção desses itens, pois eles serão o primeiro contato com o usuário.

Figura 12.11: Upload do .apk

Figura 12.12: Informações adicionais

Outros dados também devem ser informados; siga as orientações contidas na própria página para o seu preenchimento correto.

Pronto! Agora é só clicar em publicar e aguardar o processo de publicação, que geralmente leva poucas horas para ser concluído.

CAPÍTULO 13

CONHEÇA AS NOVIDADES DO ANDROID 5.0

Uma das versões mais aguardadas do Android foi, sem dúvida, a versão 5.0, codinome Lollipop. Essa nova versão, lançada em outubro de 2014, trouxe um visual totalmente novo, batizado de *Material Design*. O sistema de notificações também recebeu alterações significativas tanto em termos estruturais como visuais e de funcionalidade. Novos widgets foram incluídos, como a `RecyclerView` e a `CardView` (disponíveis nas bibliotecas de suporte a partir da revisão 21), além de *drawables* vetoriais e animações customizadas que vão facilitar a aplicação do *Material Design*.

Ademais, o Android conta agora com um novo *runtime*, o ART, construído para melhorar o desempenho e suportar uma mescla de compilação *ahead-of-time* (AOT), *just-in-time* (JIT) e código interpretado. O suporte a arquiteturas de 64 bits também foi adicionado. No campo da conectividade, o Android 5.0 adicionou novas APIs para trabalhar com *Blutetooth Low Energy* (BLE), permitindo tanto o mecanismo de *scanning* como o de *advertising*. Dessa maneira, devemos ver em breve dispositivos Android atuando como *beacons*. A versão 5.0 também adicionou suporte a novos tipos de sensores, como os de *tilt* e os de batimento cardíaco, e ainda trouxe uma `WebView` atualizada, baseada no Chromium M37. Veja mais detalhes nas seções seguintes.

13.1 ENTENDA O MATERIAL DESIGN

O Google criou o *Material Design* para ser uma linguagem visual para os seus usuários que sintetiza princípios clássicos de um bom design, levando em conta a inovação e as possibilidades da ciência e tecnologia. Basicamente, o *Material Design* considera que os materiais (elementos do design) são feitos de papel e tinta. O objetivo é criar um esquema visual único que permita unificar a experiência do usuário entre diferentes plataformas e dispositivos de diferentes tamanhos. Para isso, existem padrões de ícones, cores, animações, tipografia e hierarquias.

O *Material Design* não é, portanto, algo exclusivo do Android. Veremos esse tipo de design em outros produtos do Google, como o Chrome OS, por exemplo. Ele é composto por alguns princípios:

- **Material como metáfora —** Os materiais são a base do design, unificando, assim, o espaço e o sistema de movimentos. O material representa, por meio de sua superfície e bordas, uma realidade que se pode tocar. Os recursos de iluminação e sombras, de superfície e de movimento são elementos principais para demonstrar como os objetos se movem, interagem e coexistem em um mesmo espaço. Uma iluminação realística divide os espaços e indica partes que podem se mover.

- **Visual chamativo, gráfico e intencional —** Os elementos fundamentais como tipografia, *grids*, espaço, escala e cor promovem a criação de hierarquias, significados e foco. A escolha das cores vibrantes, imagens ocupando melhor o espaço disponível e textos com tamanho de fonte maior, bem como o uso intencional de espaços em branco, proporcionam um visual mais gráfico e chamativo. A ênfase nas ações do

usuário tornam as funcionalidades principais da aplicação fáceis de descobrir e de iniciar o uso.

- **Movimentos tem significado** — O usuário é o responsável por iniciar todo e qualquer movimento. As ações realizadas por ele são pontos que iniciam o movimento e transformam todo o design. Apesar dessas transformações, os objetos são apresentados para o usuário mantendo a continuidade da experiência de uso. Os movimentos têm significado e servem para focar sua atenção, com transições eficientes e coerentes.

O Android 5.0 conta com dois novos temas baseados no *Material Design*, como mostra a figura adiante. Também temos o `RecyclerView` como uma alternativa mais flexível em relação a `ListView` e o `CardView` , que permite a criação de interfaces gráficas baseadas em cartões. A partir de agora, as *views* do Android também possuem um atributo que indica a sua posição no eixo Z, definindo a sua elevação. *Views* com maior valor no eixo Z aparecem sobre as com valor menor.

Quanto maior a elevação de uma *view*, maior deve ser a sombra gerada por ela. Para se aprofundar em todos os aspectos do *Material Design* e obter as recomendações para os principais elementos de interface gráfica, não deixe de acessar http://www.google.com/design/spec.

Figura 13.1: Novos temas do Android 5.0. Fonte: developer.android.com

13.2 OBTENHA MAIS FLEXIBILIDADE E DESEMPENHO COM A RECYCLERVIEW

Um dos widgets mais utilizados do Android é a `ListView`. No entanto, quando precisamos criar *views* de uma forma dinâmica, ou ainda fazer um ajuste fino no desempenho da listagem — quando esta possui muitos elementos —, o uso da `ListView` exige mais cuidado e geralmente é necessário empregar o padrão *View Holder*. Esse padrão consiste basicamente em criar uma classe (o *holder*) para armazenar uma referência dos elementos da tela (como `TextViews`, `ImageViews`) diretamente no objeto que representa o layout (um `LinearLayout`, por exemplo), usando a propriedade `tag`.

O `RecyclerView` já trabalha com o padrão *View Holder* e traz a possibilidade de trabalhar com gerenciadores de layout diferentes, permitindo a criação de listas (como a `ListView`) e de *grids*, como também a renderização do layout na ordem reversa e a configuração da orientação de exibição (horizontal ou vertical). Por conta da maior flexibilidade, o `RecyclerView` acaba sendo um pouco mais complexo de se utilizar.

Assim como na `ListView`, é necessário criar um `Adapter` para fornecer o conteúdo que será exibido. A `RecyclerView` está

disponível como uma biblioteca de suporte v7; portanto, precisamos adicioná-la como dependência do nosso projeto para conseguirmos usá-la. Importe o projeto `recyclerview` , existente no diretório `sdk > extras > android > support > v7` , configure-a como um projeto de biblioteca e adicione-a como biblioteca no projeto em que deseja usá-la. Instruções detalhadas de como fazer isso encontram-se no capítulo *Desenvolva para todas as versões do Android*.

Para experimentar esse novo widget, criaremos uma `activity` que exibirá um *grid* com os livros disponíveis no nosso catálogo. A figura a seguir mostra o resultado esperado. Cada livro terá uma imagem, um título e um autor que devem ser exibidos. Para representar esse domínio, crie uma classe `Book` , dessa forma:

Figura 13.2: Grid com RecyclerView

```java
public class Book {

    private String title;
    private String author;
    private int image;

    public Book(String title, String author, int image) {
        this.title = title;
        this.author = author;
        this.image = image;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public int getImage() {
        return image;
    }

    public void setImage(int image) {
        this.image = image;
    }
}
```

Crie também uma nova `activity`, chamada `GridActivity`, e um arquivo de layout com nome de `activity_grid`. O nosso layout será composto apenas de uma `FrameLayout` e do novo widget `RecyclerView`, usando o valor `grid` como identificador. O código a seguir mostra como ficou o XML:

```xml
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent" >

    <android.support.v7.widget.RecyclerView
        android:id="@+id/grid"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</FrameLayout>
```

Cada item do *grid* será exibido com um imagem, representando o livro, o título e o nome do autor. O layout definido no arquivo `grid_item.xml` , que representa esses itens, pode ser conferido na sequência:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical"
    android:padding="4dp" >

    <ImageView
        android:id="@+id/image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp" />

    <TextView
        android:id="@+id/author"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="12sp" />

</LinearLayout>
```

Quando criamos um `RecyclerView` , precisamos configurar qual tipo de gerenciador de layout utilizaremos e um *adapter* que fornecerá o conteúdo. O *adapter* trabalha com o padrão *View*

Holder; portanto, vamos criar uma classe, estendendo uma classe base já disponibilizada, para implementá-lo. Tudo isso dentro da `GridActivity`, acompanhe a implementação:

```
public class GridActivity extends ActionBarActivity{

    private RecyclerView grid;
    private List<Book> books;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_grid);
    }

    public static class GridViewHolder
            extends RecyclerView.ViewHolder {

        protected ImageView image;
        protected TextView title;
        protected TextView author;

        private GridViewHolder(View view) {
            super(view);
            image = (ImageView) view.findViewById(R.id.image);
            title = (TextView) view.findViewById(R.id.title);
            author = (TextView) view.findViewById(R.id.author);
        }
    }
}
```

Repare que o nosso `GridViewHolder` estende uma classe base já disponibilizada pela `RecyclerView`; o que fizemos foi basicamente armazenar uma referência para os elementos de layout que representam cada item do *grid*. A vantagem de se usar esse padrão é que o número de invocações do `findViewById` cai consideravelmente, melhorando o desempenho, já que alguns ciclos de processamento são poupados.

Agora que já temos o nosso próprio *View Holder*, podemos criar o *adapter* que fará uso dele. Vamos estender novamente uma classe base já disponibilizada e implementar três métodos. O primeiro é

`getItemCount` , que devolve a quantidade de elementos existentes; o segundo é o `onCreateViewHolder` , chamado quando é necessário criar um novo item; e, por último, o `onBindViewHolder` , invocado quando um item precisa ser exibido. Além disso, nosso *adapter* recebe no construtor a lista de livros que será mostrada. Veja como ficou a implementação:

```
public class GridActivity extends ActionBarActivity{

    ...

    public static class GridAdapter extends
            RecyclerView.Adapter<GridViewHolder> {

        private List<Book> books;

        public GridAdapter(List<Book> books) {
            this.books = books;
        }

        @Override
        public int getItemCount() {
            return books.size();
        }

        @Override
        public GridViewHolder onCreateViewHolder(
        ViewGroup parent, int viewType) {
            View view = LayoutInflater
                            .from(parent.getContext())
                            .inflate(R.layout.grid_item,
                                    parent, false);

            GridViewHolder holder = new GridViewHolder(view);
            return holder;
        }

        @Override
        public void onBindViewHolder(
        GridViewHolder viewHolder, int position) {
            Book book = books.get(position);
            viewHolder.image.setImageResource(book.getImage());
            viewHolder.title.setText(book.getTitle());
            viewHolder.author.setText(book.getAuthor());
        }
    }
```

```
}
```

Para finalizar a construção da grade de livros, falta definir o tipo de layout que vamos utilizar e também associar o *adapter*. Existem três gerenciadores de layout que podem ser usados: o `GridLayoutManager`, que gera o conteúdo em formato de grade; o `StaggeredGridLayoutManager`, que permite a criação de *grids*, cuja determinada posição pode ser um item vazio; e o `LinearLayoutManager`, que permite a criação de listas.

Utilizaremos o `GridLayoutManager` em nosso exemplo. Criar um novo gerenciador de layout e associar o *adapter* na `RecyclerView` é a parte fácil, veja no código a seguir:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_grid);

    grid = (RecyclerView) findViewById(R.id.grid);

    //grid with 3 columns
    LayoutManager layoutManager = new GridLayoutManager(this,3);
    grid.setLayoutManager(layoutManager);

    GridAdapater adapter = new GridAdapter(createBooks());
    grid.setAdapter(adapter);
}
private List<Book> createBooks() {
    books = new ArrayList<Book>();
    for (int i = 1; i <= 30; i++) {
        Book book = new Book("Title " + i, "Author " + i,
                                R.drawable.ic_cover);
        books.add(book);
    }
    return books;
}
```

Ao executar o aplicativo, obteremos o resultado apresentado na figura *Grid com RecyclerView*. Para deixar o exemplo mais completo, precisamos fazer com que a `RecyclerView` responda à seleção realizada pelo usuário. Aqui não temos algo como o

`OnItemClickListener` . Para responder às ações do usuário, vamos recorrer ao mecanismo de detecção de gestos.

O que faremos inicialmente é estender a classe `SimpleOnGestureListener` , que já nos fornece uma detecção básica de gestos, e sobrescreveremos o método `onSingleTapConfirmed` , que indica que o usuário tocou em determinado item da grade. Analise o código a seguir:

```
public class GridActivity extends ActionBarActivity{

...

    private class GridGestureDetector
                    extends SimpleOnGestureListener {

        @Override
        public boolean onSingleTapConfirmed(MotionEvent e) {
            View v = grid.findChildViewUnder(e.getX(), e.getY());
            int i = grid.getChildPosition(v);
            Book book = books.get(i);
            Toast.makeText(GridActivity.this, "
                            Book " + book.getTitle(),
                            Toast.LENGTH_SHORT)    .show();
            return super.onSingleTapConfirmed(e);
        }
    }
}
```

No método `onSingleTapConfirmed` , verificamos qual *view* se encontra nas coordenadas X e Y do evento de toque disparado pelo usuário, e posteriormente buscamos qual é a posição na grade do item que foi selecionado. Uma vez que temos a posição do item, basta obtê-lo da nossa lista de livros. Agora, vamos conectar o detector de gestos na `RecyclerView` . Para isso, faremos a `GridActivity` implementar um *listener* que delegará o tratamento do evento para o detector de gestos. Veja o código:

```
public class GridActivity extends ActionBarActivity
                            implements OnItemTouchListener {
    ...
```

```java
    private GestureDetectorCompat detector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...

        detector = new GestureDetectorCompat(
                                this, new GridGestureDetector());
        grid.addOnItemTouchListener(this);
    }

    @Override
    public boolean onInterceptTouchEvent(
    RecyclerView recyclerView, MotionEvent motionEvent) {
        detector.onTouchEvent(motionEvent);
        return false;
    }

    @Override
    public void onTouchEvent(RecyclerView view,
                                MotionEvent event) {}

}
```

No método `onCreate`, criamos o detector de gestos e passamos a `activity` como implementação do `addOnItemTouchListener` do `grid`. No método `onInterceptTouchEvent`, que intercepta os eventos de toque realizados no `grid`, delegamos o tratamento do evento para o nosso detector de gestos. Agora nosso exemplo está completo!

13.3 CRIE LAYOUTS BASEADO EM CARTÕES

O `CardView` é um novo widget que facilita a criação de cartões e que tem sido muito utilizado em vários aplicativos, principalmente nos do Google com a Play Store. A figura a seguir mostra uma exemplo de uso de cartões. De acordo com o *Material Design*, um cartão representa um pedaço de papel que contém um conjunto de informações de um determinado assunto, geralmente funcionando como um ponto de partida para acessar informações, para assim detalhadas.

Figura 13.3: Exemplo de uso de cartões

Os cartões têm largura fixa, altura variável e devem ter as bordas arredondadas. Eles são uma boa maneira de exibir conteúdo formado por diferentes tipos de elementos, como imagens, textos, links e botões de ação. Para saber mais sobre em quais situações utilizá-los, consulte http://www.google.com/design/spec/components/cards.html#cards-usage.

O `CardView` está disponível via biblioteca de suporte v7 em um projeto separado como a `RecyclerView`. O procedimento para adicioná-lo ao seu projeto é o mesmo descrito para a `RecyclerView`. Para usar o `CardView`, basta incluir no seu arquivo de layout o seguinte código:

```
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    card_view:cardCornerRadius="4dp" >

    <!-- card elements here-->

</android.support.v7.widget.CardView>
```

Podemos utilizar o `CardView` juntamente com a `RecyclerView`, no qual cada elemento que deve ser exibido é um cartão. Para experimentar a `CardView`, altere o layout do arquivo `grid_item.xml` para incluir este novo widget, dessa forma:

```
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    card_view:cardCornerRadius="4dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal" >

        <ImageView
            android:id="@+id/image"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >

            <TextView
                android:id="@+id/title"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="32sp" />

            <TextView
```

```
                android:id="@+id/author"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="24sp" />
        </LinearLayout>
    </LinearLayout>

</android.support.v7.widget.CardView>
```

Também vamos alterar o tipo de layout do `RecyclerView` para ser uma lista. Para isso, basta trocar o `GridLayoutManager` pelo `LinearLayoutManager`, como mostra o código a seguir:

```
//GridActivity
protected void onCreate(Bundle savedInstanceState) {
    ...
    LayoutManager layoutManager = new LinearLayoutManager(this);
    ...
```

Agora é só executar o aplicativo novamente e o resultado obtido deverá ser similar ao apresentado na figura:

Figura 13.4: Uso combinado de RecyclerView e CardView

13.4 PROMOVA AÇÕES COM BOTÕES FLUTUANTES

Com o *Material Design* também houve alterações quanto a forma de disponibilizar ações para o usuário. Um botão é uma maneira (com texto e/ou imagem) de comunicar ao usuário sobre qual ação acontecerá quando ele for pressionado. O *Material Design* define três tipos de botão:

- ***Flat button*** **—** é uma representação textual, feita de tinta, que reage ao ser pressionado, mudando de cor.
- ***Raised button*** **—** é um botão retangular, feito de papel, que tem efeito de elevação e mudança de cor, quando

pressionado. É muito semelhante a um botão tradicional.

- ***Floating action button* —** é um botão circular também feito de papel e que apresenta efeitos de mudança de cor e elevação ao ser pressionado.

Figura 13.5: Tipos de botão. Fonte: google.com/design

Essa figura mostra os três tipos de botões. Os botões de ação flutuantes devem ser usados quando a ação representada por ele é importante e faz sentido ser exibida em conjunto com qualquer dado existente na tela. Como essas são ações visíveis diretamente na interface gráfica, e não com um item da `ActionBar` , elas são geralmente chamadas de *promoted actions*. No novo aplicativo do Gmail, por exemplo, a ação de criar um novo e-mail é uma *promoted action*, como pode ser visto na figura a seguir.

Figura 13.6: Botão de ação flutuante no Gmail

No nosso catálogo poderíamos incluir uma *promoted action* para a ação de *checkout* do pedido. O Android 5.0 trouxe facilidades para criar esse tipo de botão; porém, não é retrocompatível com versões anteriores. Felizmente, já existem projetos da comunidade que implementam esse recurso e o disponibiliza para várias versões. Além disso, na versão 22.2.0 das bibliotecas de suporte, foi incluída uma biblioteca de design que contém uma implementação dos botões de ação flutuantes.

Vamos utilizar essa biblioteca (https://github.com/futuresimple/android-floating-action-button) para fazer nossa implementação. Ela é disponibilizada via Gradle; portanto, basta adicionar a seguinte dependência no arquivo

`build.gradle`:

```
dependencies {
    compile 'com.getbase:floatingactionbutton:1.9.1'
}
```

Para começar a usar a nova biblioteca, no arquivo `grid_activity` adicionaremos o botão de ação flutuante. Para facilitar o seu posicionamento, vamos substituir o `FrameLayout` por um `RelativeLayout`, dessa forma:

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:fab="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <android.support.v7.widget.RecyclerView
        android:id="@+id/grid"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="8dp" />

    <com.getbase.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_marginBottom="16dp"
        fab:fab_colorNormal="@color/button_material_dark"
        fab:fab_colorPressed="@color/button_material_light"
        fab:fab_icon="@drawable/ic_action_cart" />

</RelativeLayout>
```

Ao executar a aplicação novamente, você terá um botão flutuante no canto inferior esquerdo da tela, com o ícone de *checkout*, como mostra a figura seguinte. Para saber mais como usar os tipos de botão, visite http://www.google.com/design/spec/components/buttons.html#buttons-usage.

Figura 13.7: Botão de ação flutuante no Catálogo

CAPÍTULO 14

CONTINUE OS ESTUDOS

Ufa! Depois de uma longa e boa jornada, chegamos ao final do nosso livro. Criamos uma aplicação cheia de funcionalidades interessantes e outros exemplos que poderão servir de base ou inspiração para que você desenvolva seus projetos.

No entanto, é importante saber que seus estudos não podem parar por aqui. Existe muito material bom mundo afora. A primeira fonte onde você deve sempre buscar por informações é a página oficial do Android, http://developer.android.com.

Fique atento e acompanhe o lançamento das novas versões do Android, que sempre vêm com novidades, não só para o usuário final, mas também para o desenvolvedor. São publicados na página oficial diversos guias que mostram como funcionam os novos recursos. Existem também muitos livros bons, como o *Android in action*, de W. Frank Ableson, Robi Sen, Chris King e C. Enrique Ortiz.

Crie também o hábito de investigar o código fonte do Android, pois ele é sempre uma boa referência de implementação e descobertas (http://source.android.com). Outro ponto importante é conhecer e aplicar as *guidelines* de design, recomendadas pelo Google em https://developer.android.com/design/. Valorize a experiência do usuário, pois ela é um ponto chave para o sucesso da sua aplicação.

Além disso, preocupe-se com a qualidade do código produzido, e utilize ferramentas e frameworks para auxiliá-lo em tarefas corriqueiras. Alguns exemplos de frameworks que podem ajudar são:

- **ORM Lite** — ferramenta para mapeamento objeto-relacional compatível com Android (http://ormlite.com).
- **Android Query** — biblioteca com funções utilitárias para requisições e tarefas assíncronas, manipulação de `views` e outras utilidades (https://github.com/androidquery/androidquery);
- **Android Annotations** — framework que resolve situações recorrentes como *binding* de `views`, recursos do sistema e criação de `listeners` (http://androidannotations.org/);
- **Spring Android** — framework que oferece facilidades para comunicação com serviços REST e autenticação OAuth (http://www.springsource.org/spring-android);
- **http-request** — utilitário para comunicação via HTTP (https://github.com/kevinsawicki/http-request);
- **Retrofit** — cliente REST *type-safe* (http://square.github.io/retrofit/);
- **Android Arsenal** — site que relaciona diversas bibliotecas e ferramentas que podem ser utilizadas no seu projeto (https://android-arsenal.com/).

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Por fim, espero que você tenha muito sucesso nas suas

aplicações Android e que continue aperfeiçoando seus conhecimentos!